Unlocking the Potential of Domain Aware Binary Analysis in the Era of IoT

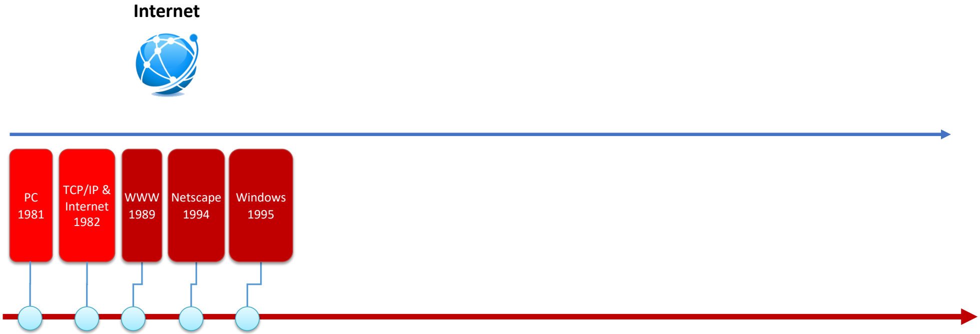
Zhiqiang Lin

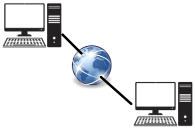
zlin@cse.ohio-state.edu

April 18th, 2023

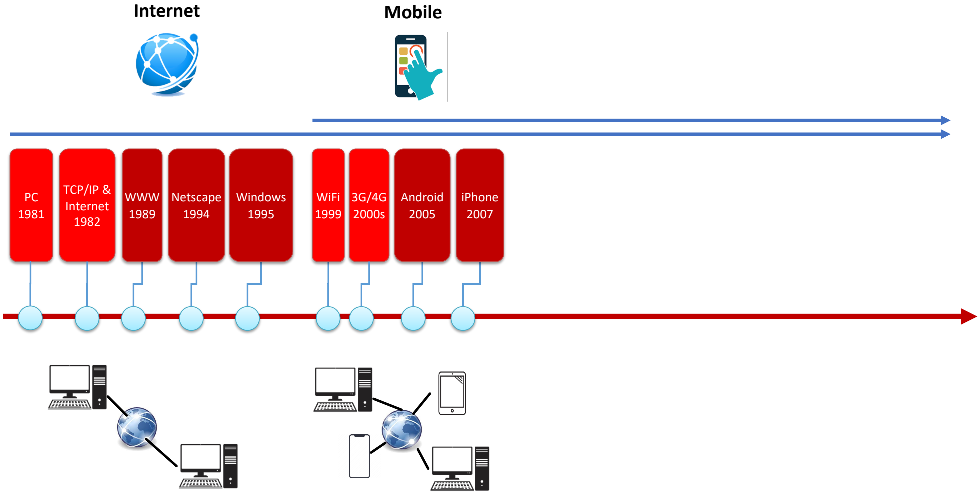
History of Computing (Since 1980s)

History of Computing (Since 1980s)

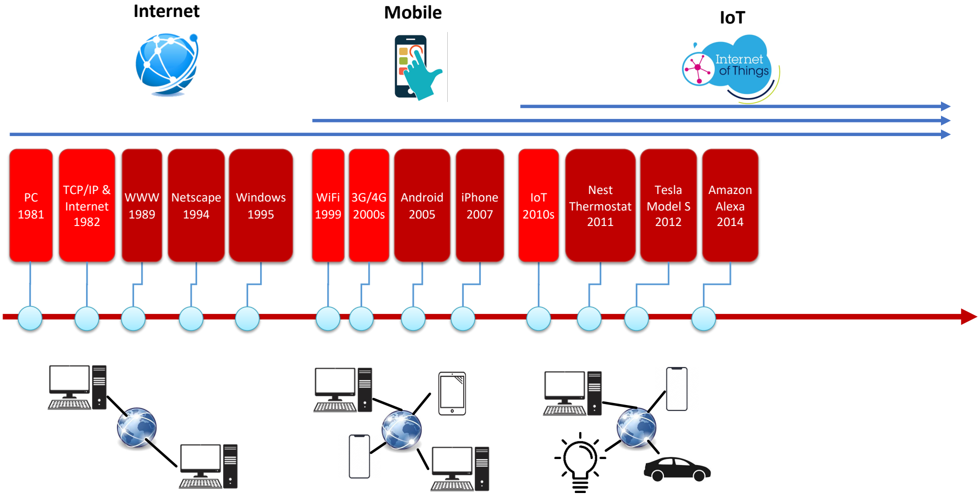




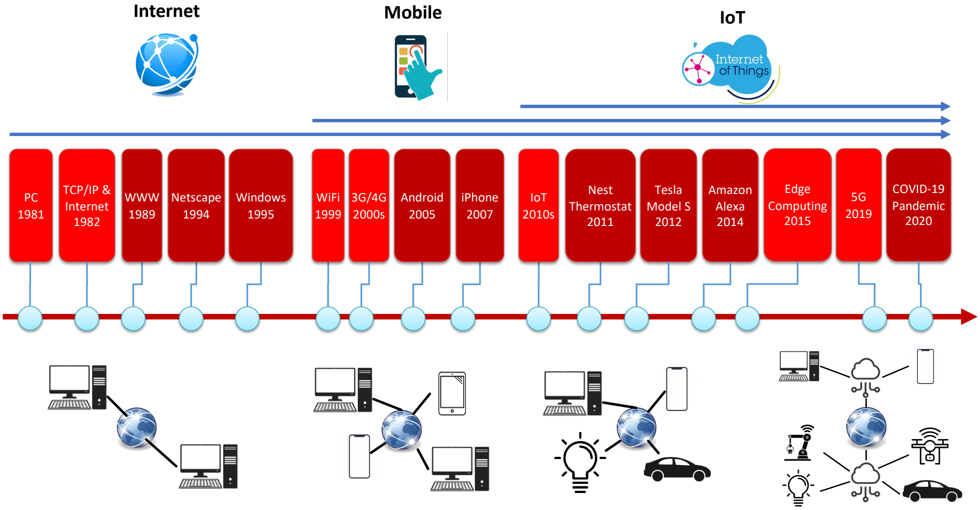
History of Computing (Since 1980s)



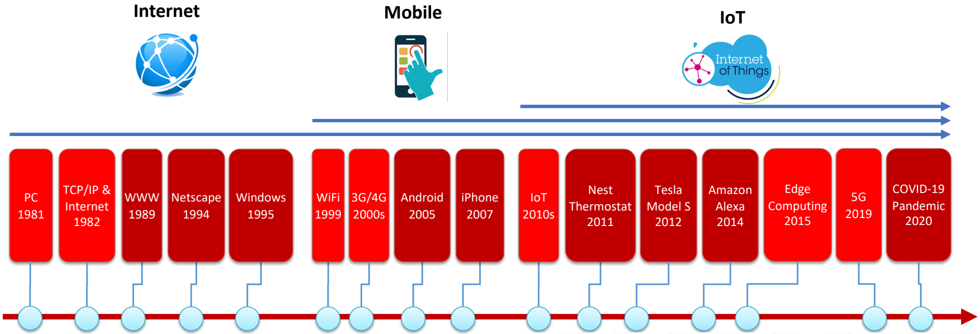
History of Computing (Since 1980s)

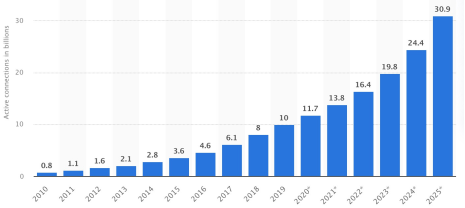


History of Computing (Since 1980s)



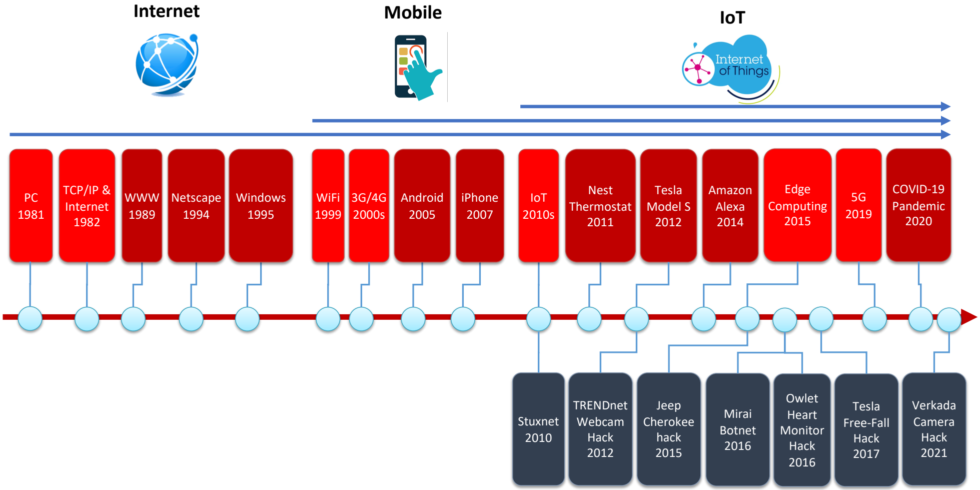
History of Computing (Since 1980s)



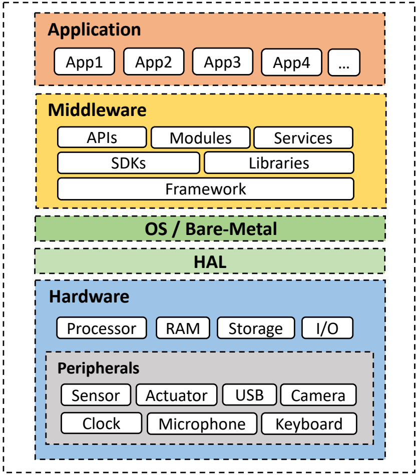


Data Source: statista

History of Computing (Since 1980s)

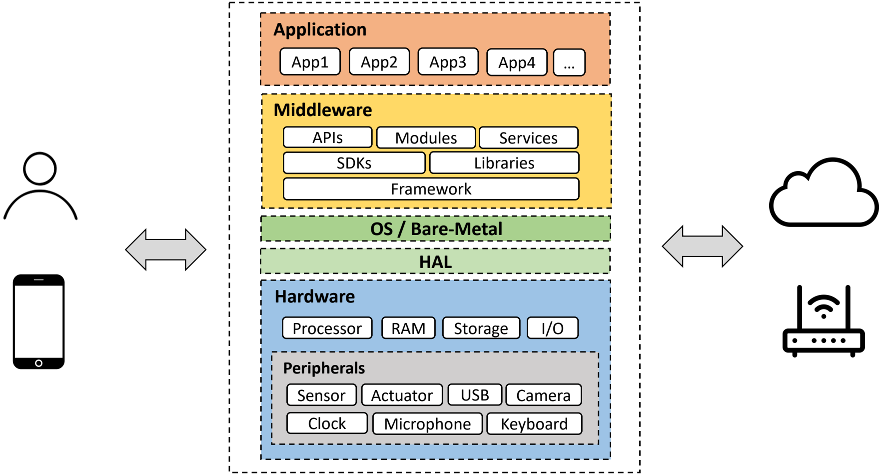


Modern IoT Architecture



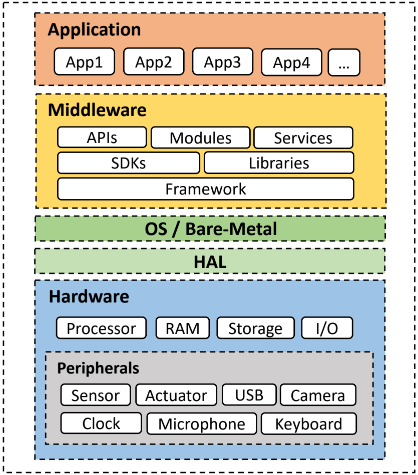
An Embedded IoT Device

Modern IoT Architecture



An Embedded IoT Device

Modern IoT Architecture



An Embedded IoT Device

Modern IoT Architecture

Application

App1 | App2 | App3 | App4 | ...

Middleware

APIs | Modules | Services

SDKs | Libraries

Framework

OS / Bare-Metal

HAL

Hardware

Processor | RAM | Storage | I/O

Peripherals

Sensor | Actuator | USB | Camera

Clock | Microphone | Keyboard

An Embedded IoT Device

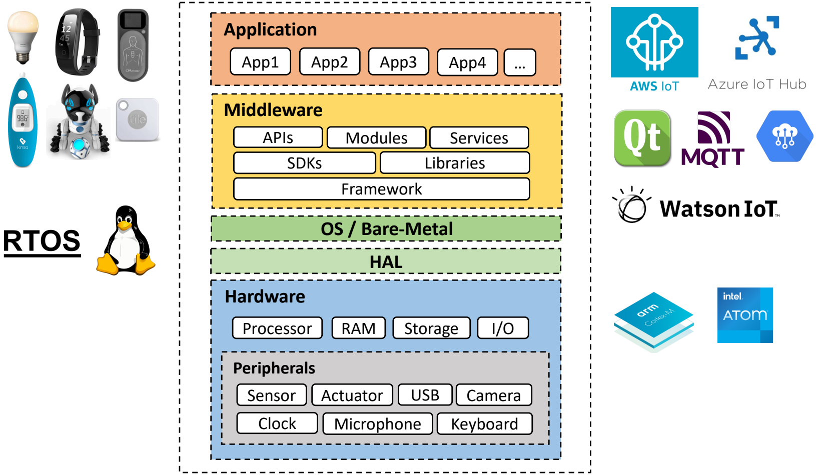
AWS IoT

Azure IoT Hub

MQTT

Watson IoT

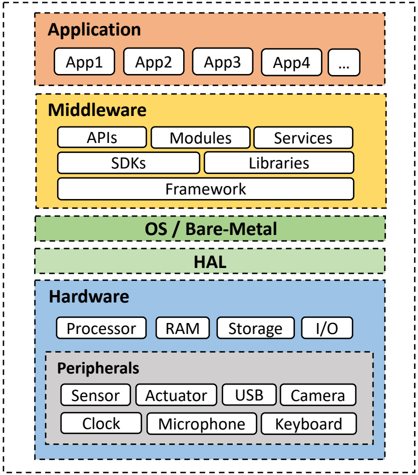
Modern IoT Architecture



An Embedded IoT Device

Modern IoT Architecture

RTOS



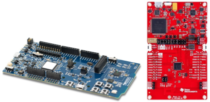
An Embedded IoT Device

AWS IoT

Azure IoT Hub

MQTT

Watson IoT



Domain-Aware Binary Analysis

Binary code analysis is *challenging*

- Control flow recovery, semantic understanding, vulnerability detection, root-cause analysis...

Domain-Aware Binary Analysis

Binary code analysis is *challenging*

- Control flow recovery, semantic understanding, vulnerability detection, root-cause analysis...

Why Domain-Aware

1. *One size does not fit all*
   - Heterogeneous architecture, OS, APIs of different IoT vendors
   - Domain-specific challenges

Domain-Aware Binary Analysis

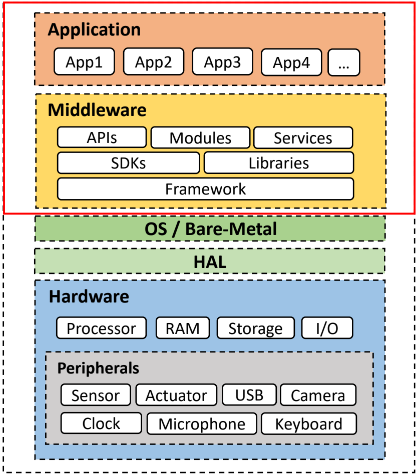
Binary code analysis is *challenging*

- Control flow recovery, semantic understanding, vulnerability detection, root-cause analysis...

Why Domain-Aware

1. *One size does not fit all*
   - Heterogeneous architecture, OS, APIs of different IoT vendors
   - Domain-specific challenges
2. *Learn from the domain*
   - Unique domain insights for binary analysis
   - Novel techniques and methodology
   - Transition to other domains

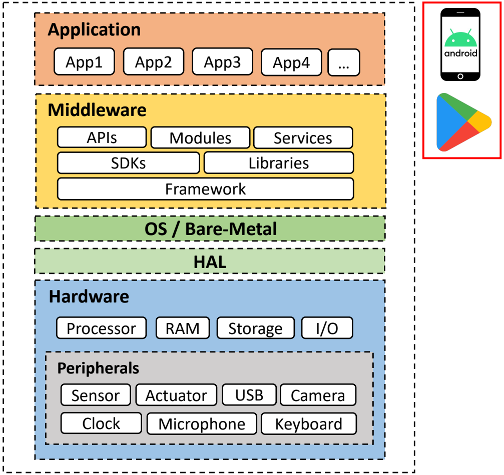
Our Recent Works on (IoT) Binary Analysis



An Embedded IoT Device

1. **Egg Hunt in Tesla Infotainment: A First Look at Reverse Engineering of Qt Binaries**. In USENIX Security 2023

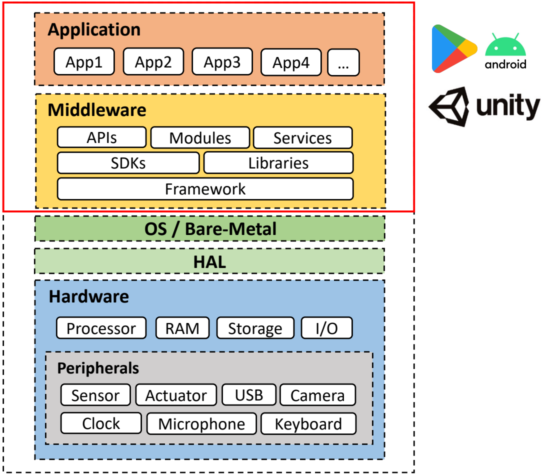
Our Recent Works on (IoT) Binary Analysis



An Embedded IoT Device

1. **Egg Hunt in Tesla Infotainment: A First Look at Reverse Engineering of Qt Binaries.** In USENIX Security 2023
2. **Understanding IoT Security from a Market-Scale Perspective.** In CCS 2022

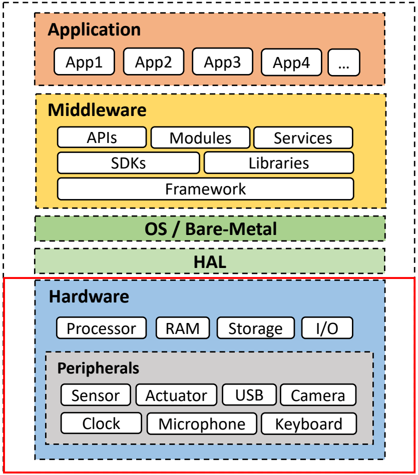
Our Recent Works on (IoT) Binary Analysis



An Embedded IoT Device

1. **Egg Hunt in Tesla Infotainment: A First Look at Reverse Engineering of Qt Binaries**. In USENIX Security 2023
2. **Understanding IoT Security from a Market-Scale Perspective**. In CCS 2022
3. **Playing Without Paying: Detecting Vulnerable Payment Verification in Native Binaries of Mobile Games**. In USENIX Security 2022

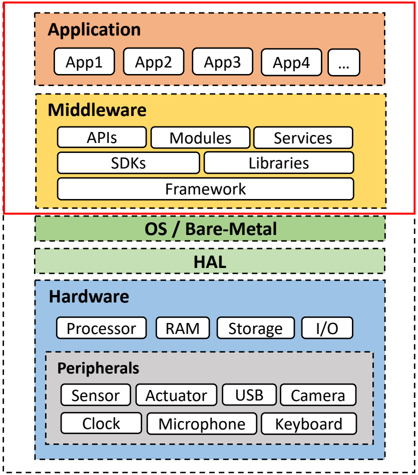
Our Recent Works on (IoT) Binary Analysis



An Embedded IoT Device

1. **Egg Hunt in Tesla Infotainment: A First Look at Reverse Engineering of Qt Binaries**. In USENIX Security 2023
2. **Understanding IoT Security from a Market-Scale Perspective**. In CCS 2022
3. **Playing Without Paying: Detecting Vulnerable Payment Verification in Native Binaries of Mobile Games**. In USENIX Security 2022
4. **What You See is Not What You Get: Revealing Hidden Memory Mapping for Peripheral Modeling**. In RAID 2022

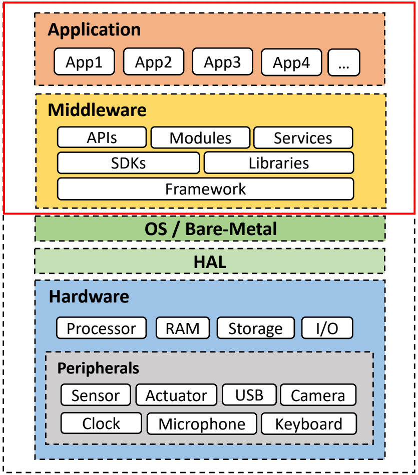
Our Recent Works on (IoT) Binary Analysis



An Embedded IoT Device

1. **Egg Hunt in Tesla Infotainment: A First Look at Reverse Engineering of Qt Binaries**. In USENIX Security 2023
2. **Understanding IoT Security from a Market-Scale Perspective**. In CCS 2022
3. **Playing Without Paying: Detecting Vulnerable Payment Verification in Native Binaries of Mobile Games**. In USENIX Security 2022
4. **What You See is Not What You Get: Revealing Hidden Memory Mapping for Peripheral Modeling**. In RAID 2022
5. **FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities from Bare-Metal Firmware**. In CCS 2020

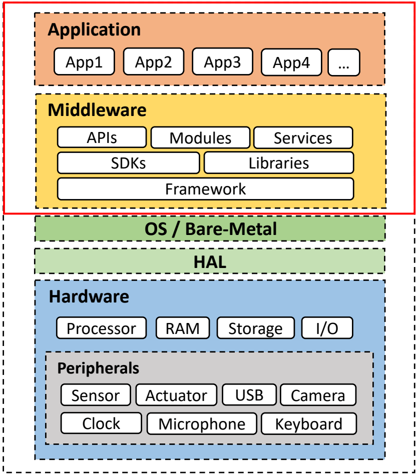
Our Recent Works on (IoT) Binary Analysis



An Embedded IoT Device

1. **Egg Hunt in Tesla Infotainment: A First Look at Reverse Engineering of Qt Binaries**. In USENIX Security 2023
2. **Understanding IoT Security from a Market-Scale Perspective**. In CCS 2022
3. **Playing Without Paying: Detecting Vulnerable Payment Verification in Native Binaries of Mobile Games**. In USENIX Security 2022
4. **What You See is Not What You Get: Revealing Hidden Memory Mapping for Peripheral Modeling**. In RAID 2022
5. **FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities from Bare-Metal Firmware**. In CCS 2020
6. **Plug-N-Pwned: Comprehensive Vulnerability Analysis of OBD-II Dongles as A New Over-the-Air Attack Surface in Automotive IoT**. In USENIX Security 2020

Our Recent Works on (IoT) Binary Analysis

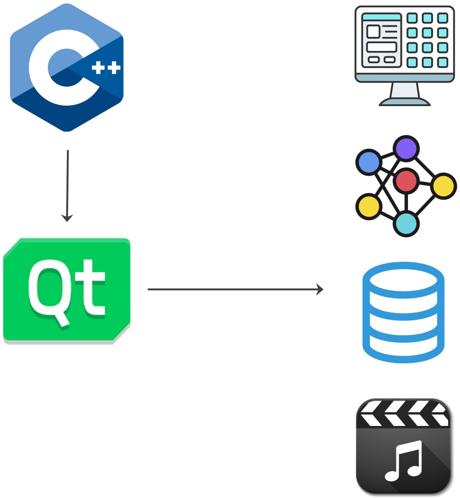


An Embedded IoT Device

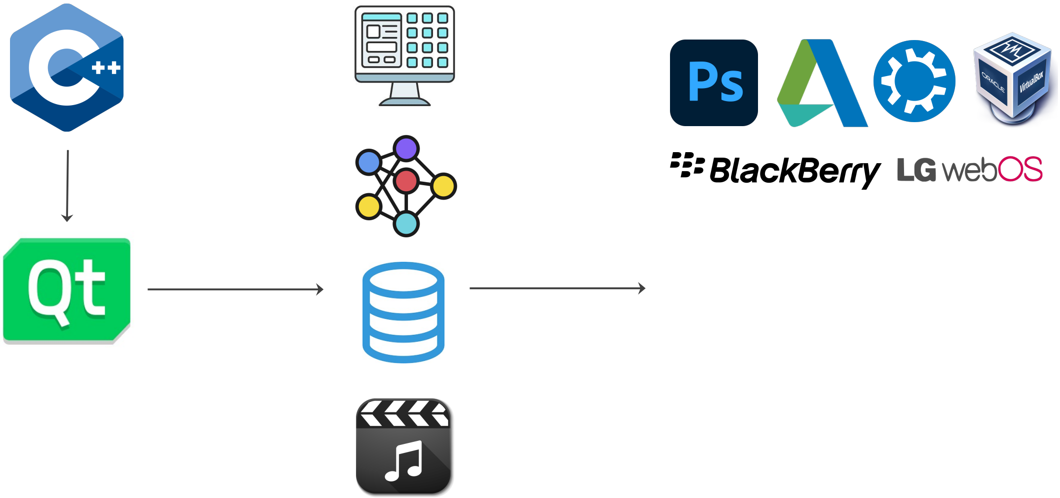
1. **Egg Hunt in Tesla Infotainment: A First Look at Reverse Engineering of Qt Binaries**. In USENIX Security 2023
2. ~~Understanding IoT Security from a Market-Scale Perspective. In CCS 2022~~
3. ~~Playing Without Paying: Detecting Vulnerable Payment Verification in Native Binaries of Mobile Games. In USENIX Security 2022~~
4. **What You See is Not What You Get: Revealing Hidden Memory Mapping for Peripheral Modeling**. In RAID 2022
5. **FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities from Bare-Metal Firmware**. In CCS 2020
6. ~~Plug-N-Pwned: Comprehensive Vulnerability Analysis of OBD-II Dongles as A New Over-the-Air Attack Surface in Automotive IoT. In USENIX Security 2020~~

Background

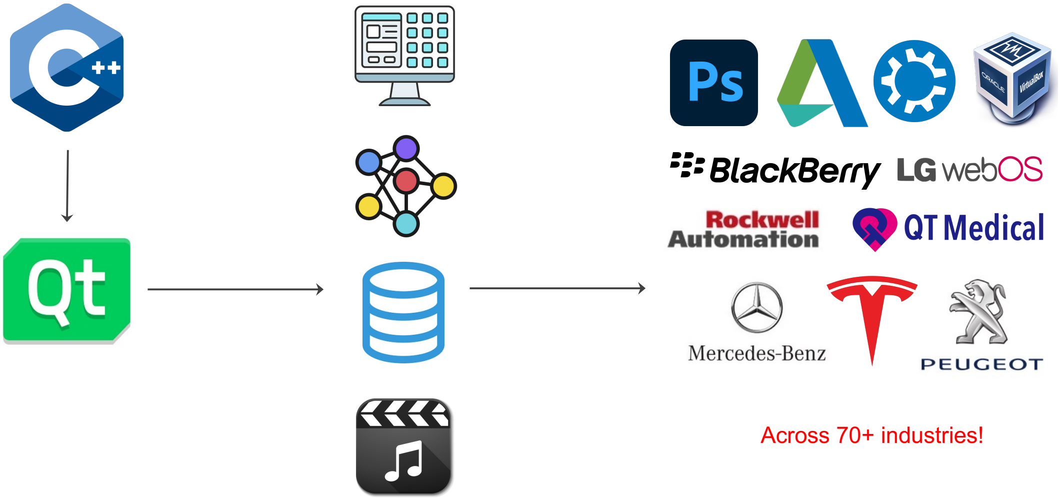
Background



Background

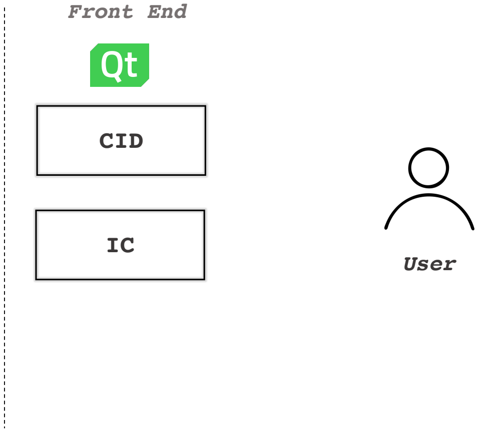


Background

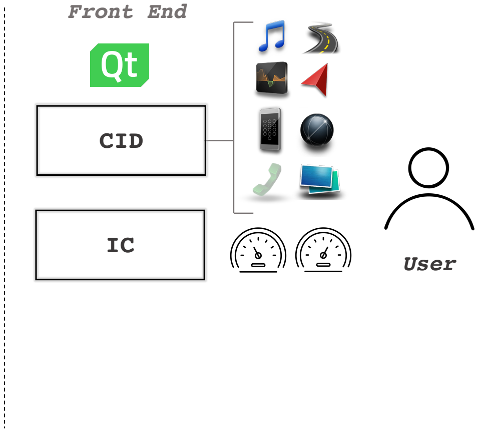


Across 70+ industries!

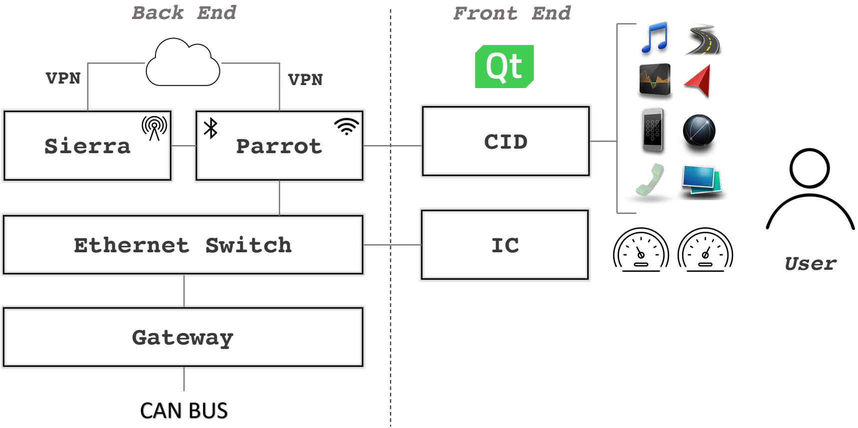
Tesla's Infotainment System



Tesla's Infotainment System



Tesla's Infotainment System



Motivation

Enabling Security Analysis of Qt Programs

- Reverse engineering (RE) is one of the keys to vet Qt binaries

Motivation

Enabling Security Analysis of Qt Programs

- Reverse engineering (RE) is one of the keys to vet Qt binaries
- Existing C++ binary analysis tools can be applied [ghi, SWS+16]

Motivation

Enabling Security Analysis of Qt Programs

- Reverse engineering (RE) is one of the keys to vet Qt binaries
- Existing C++ binary analysis tools can be applied [ghi, SWS+16]

Binary RE Challenges

- **Control Flow Graph (CFG) Recovery**. Indirect control flow transfers such as callbacks and indirect calls [PCvdV+17, VDVGC+16]

Motivation

Enabling Security Analysis of Qt Programs

- Reverse engineering (RE) is one of the keys to vet Qt binaries
- Existing C++ binary analysis tools can be applied [ghi, SWS+16]

Binary RE Challenges

- **Control Flow Graph (CFG) Recovery**. Indirect control flow transfers such as callbacks and indirect calls [PCvdV+17, VDVGC+16]
- **Symbol Recovery** (e.g., names/types of functions/variables). Code stripping during binary compilation [TTN+19, SCD+18]

Key Insights

Unique Insights from Qt's Mechanisms

Key Insights

Unique Insights from Qt's Mechanisms

1. Qt's Signal and Slot

Key Insights

Unique Insights from Qt's Mechanisms

1. Qt's Signal and Slot
   - Originally designed for efficient function callback implementation among GUIs

Key Insights

Unique Insights from Qt's #Mechanisms

1. Qt's Signal and Slot
   - Originally designed for efficient function callback implementation among GUIs
   - *We instead leverage it to identify Qt-specific function callbacks*

Key Insights

Unique Insights from Qt's Mechanisms

1. Qt's Signal and Slot
   - Originally designed for efficient function callback implementation among GUIs
   - *We instead leverage it to identify Qt-specific function callbacks*
2. Qt's Dynamic Introspection

Key Insights

Unique Insights from Qt's Mechanisms

1. Qt's Signal and Slot
   - Originally designed for efficient function callback implementation among GUIs
   - *We instead leverage it to identify Qt-specific function callbacks*
2. Qt's Dynamic Introspection
   - Originally designed for run-time class member query and update

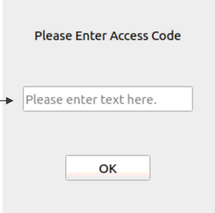
Key Insights

Unique Insights from Qt's Mechanisms

1. Qt's Signal and Slot
   - Originally designed for efficient function callback implementation among GUIs
   - *We instead leverage it to identify Qt-specific function callbacks*
2. Qt's Dynamic Introspection
   - Originally designed for run-time class member query and update
   - *We repurpose it to recover rich semantic symbols from the binary program*

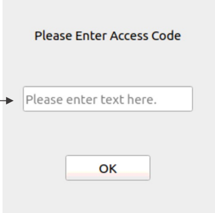
Qt's Signal and Slot Mechanism

```
MainWindow::MainWindow() {
  ...
  // Create lineEdit instance
  v0 = operator.new(0x30)
  QLineEdit(v0)
  *(this + 0x30) = v0
  ...







}
```



Qt's Signal and Slot Mechanism

```
MainWindow::MainWindow() {
  ...
  // Create lineEdit instance
  v0 = operator.new(0x30)
  QLineEdit(v0)
  *(this + 0x30) = v0
  ...
  // Register callbacks
  connect(*(this+0x30),“2textChanged(QString)”
          , this, “1updateText(QString)”, 0)

  connect(*(this+0x30),“2editingFinished()”
          , this, “1handleInput()”, 0)
  ...
}
```

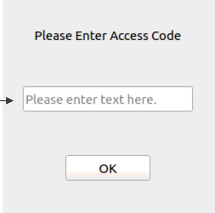


Qt’s Signal and Slot Mechanism

```
MainWindow::MainWindow() {
   ...
   // Create lineEdit instance
   v0 = operator.new(0x30)
   QLineEdit(v0)
   *(this + 0x30) = v0
   ...
   // Register callbacks
   connect(*(this+0x30),“2textChanged(QString)”
             , this, “1updateText(QString)”, 0)

   connect(*(this+0x30),“2editingFinished()”
             , this, “1handleInput()”, 0)
   ...
}
```

Signal

Slot

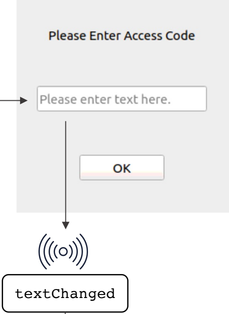


Qt's Signal and Slot Mechanism

```
MainWindow::MainWindow() {
   ...
   // Create lineEdit instance
   v0 = operator.new(0x30)
   QLineEdit(v0)
   *(this + 0x30) = v0
   ...
   // Register callbacks
   connect(*(this+0x30),"2textChanged(QString)"
            , this, "1updateText(QString)", 0)

   connect(*(this+0x30),"2editingFinished()"
            , this, "1handleInput()", 0)
   ...
}
```

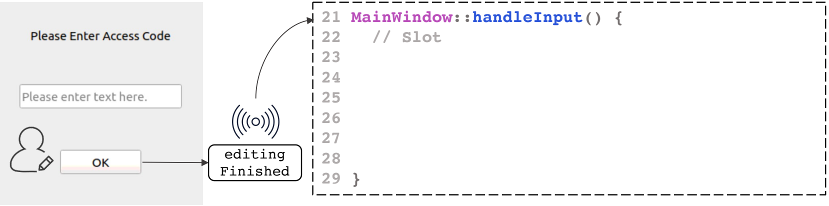
Qt's Signal and Slot Mechanism

```
MainWindow::MainWindow() {
  ...
  // Create lineEdit instance
  v0 = operator.new(0x30)
  QLineEdit(v0)
  *(this + 0x30) = v0
  ...
  // Register callbacks
  connect(*(this+0x30),"2textChanged(QString)"
          , this, "1updateText(QString)", 0)

  connect(*(this+0x30),"2editingFinished()"
          , this, "1handleInput()", 0)
  ...
}
```



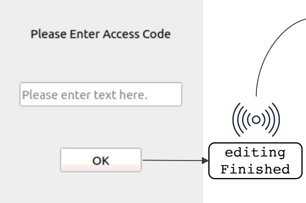

```
MainWindow::updateText(QString v1) {
  // Slot
  if (v1 != null)
      *(this + 0x48) = v1 // this->text
}
```

Qt's Dynamic Introspection



window.text = "secret"

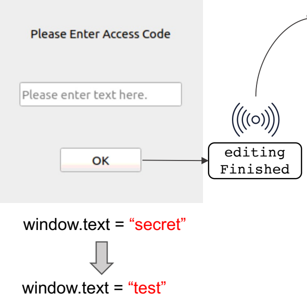
Qt's Dynamic Introspection



window.text = "secret"

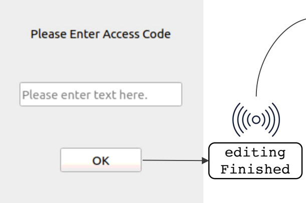
```
MainWindow::handleInput() {
  // Slot
  v1 = *(this + 0x48)     // this->text
  if (v1 == "secret") {
      // Dynamic introspection


  }
}
```

Qt's Dynamic Introspection




```
MainWindow::handleInput() {
  // Slot
  v1 = *(this + 0x48)    // this->text
  if (v1 == "secret") {
      // Dynamic introspection
      this->setProperty("text", "test")
      qDebug() << v1  // Will print out "test"
  }
}
```

Qt's Dynamic Introspection




```
MainWindow::handleInput() {
  // Slot
  v1 = *(this + 0x48)     // this->text
  if (v1 == "secret") {
     // Dynamic introspection
     this->setProperty("text", "test")
     qDebug() << v1  // Will print out "test"
  }
}
```

Query index

Metadata

Property Table

Index	Name Index	Type
0	0	QString

String Table

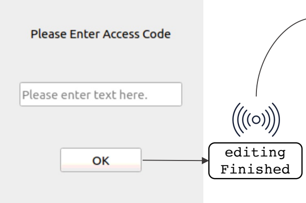
Index	String
0	text

window.text = "secret"

⇩

window.text = "test"

Qt's Dynamic Introspection



window.text = "secret"

window.text = "test"

```
MainWindow::handleInput() {
  // Slot
  v1 = *(this + 0x48)     // this->text
  if (v1 == "secret") {
     // Dynamic introspection
     this->setProperty("text", "test")
     qDebug() << v1  // Will print out "test"
  }
}
```

Query index

Metadata

Property Table

Index	Name Index	Type
0	0	QString

String Table

Index	String
0	text

```
MainWindow::qt_metacall(… int v1, void** v2) {
  ...





}
```

Qt’s Dynamic Introspection

editing Finished

window.text = “secret”

window.text = “test”

```
MainWindow::handleInput() {
  // Slot
  v1 = *(this + 0x48)     // this->text
  if (v1 == “secret”) {
      // Dynamic introspection
      this->setProperty(“text”, “test”)
      qDebug() << v1  // Will print out “test”
  }
}
```

Query index

```
MainWindow::qt_metacall(… int v1, void** v2) {
  ...
  if (v1 == 0) {
      // Set property value by index
      *(this + 0x48) = (QString) v2
  }
}
```

Metadata

Property Table

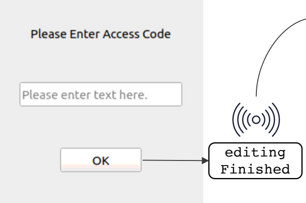
Index	Name Index	Type
0	0	QString

String Table

Index	String
0	text

Qt's Dynamic Introspection

Qt's Dynamic Introspection




```
MainWindow::handleInput() {
  // Slot
  v1 = *(this + 0x48)     // this->text
  if (v1 == "secret") {
     // Dynamic introspection
     this->setProperty("text", "test")
     qDebug() << v1  // Will print out "test"
  }
}
```

Query index

```
MainWindow::qt_metacall(… int v1, void** v2) {
  ...
  if (v1 == 0) {
     // Set property value by index
     *(this + 0x48) = (QString) v2
  }
}
```

Metadata

Property Table

Index	Name Index	Type
0	0	QString

String Table

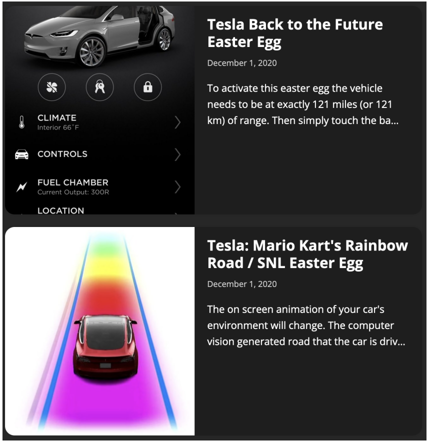
Index	String
0	text

*(this + 0x48)

Type: Qstring

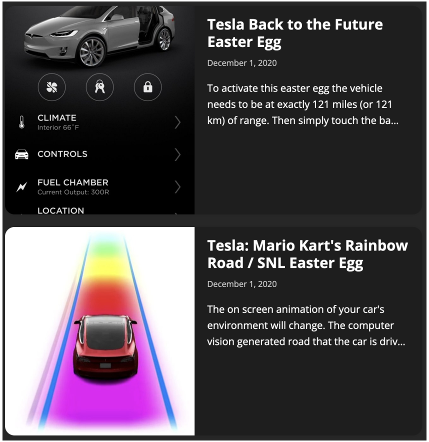
Name: text

Application: Egg Hunt in Tesla Infotainment



Easter eggs in Tesla vehicles

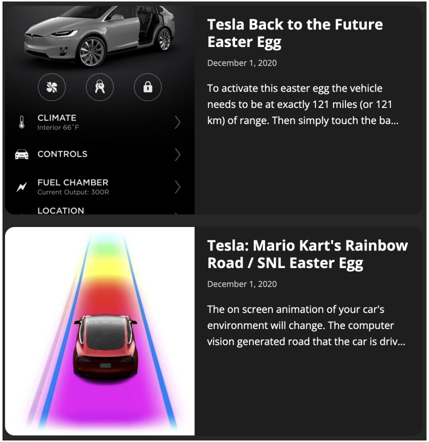
Application: Egg Hunt in Tesla Infotainment



Easter eggs in Tesla vehicles

- Do they raise security concerns?
- How to systematically identify them?

Application: Egg Hunt in Tesla Infotainment



Easter eggs in Tesla vehicles

- Do they raise security concerns?
- How to systematically identify them?
  - Coverage-based fuzzing (emulation required)
  - Input validation analysis on Qt binaries

Application: Egg Hunt in Tesla Infotainment

Experiment Setup

- Use input validation analysis to extract hidden commands from Tesla firmware
- Identify user input variables from the recovered Qt symbols
- Analyze the recovered Qt control flow

Class Name	**Var./Func. Name**
QLineEdit	text()
QLineEdit	text
QAbstractSpinBox	text
QDoubleSpinBox	text
QSpinBox	text
QDateTimeEdit	text
TextField	text
PasswordTextField	text
WebEntryField	text
NavigationSearchBox	text
CompleterTextField	text
ExtEntryField	text

Table: Identified user input variables.

Application: Egg Hunt in Tesla Infotainment

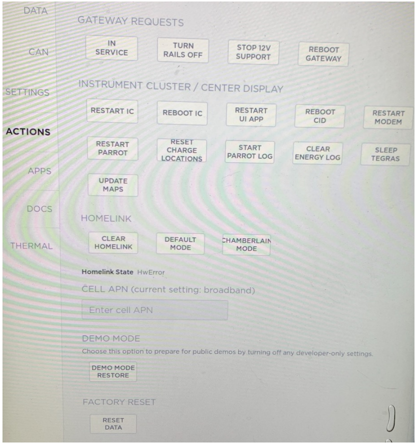
Category	Content	Description
Easter Egg	"007"	Submarine Easter egg
	"modelxmas"	Show holiday lights
	"42"	Change car name
	"mars"	Turn map into Mars surface
	"transport"	Transport mode
	"performance"	Performance mode
	"showroom"	Showroom mode
Access Token	SecurityToken1	Enable diagnostic mode
	SecurityToken2	Enable diagnostic mode
	crc(token)==0x18e5a977	Enable developer mode
	crc(token)==0x73bbee22	Enable developer mode
Master Pwd	"3500"	Exit valet mode

Table: Hidden commands from Tesla firmware.

Application: Egg Hunt in Tesla Infotainment

Category	Content	Description
	“007”	Submarine Easter egg
	“modelxmas”	Show holiday lights
Easter	“42”	Change car name
Egg	“mars”	Turn map into Mars surface
	“transport”	Transport mode
	“performance”	Performance mode
	“showroom”	Showroom mode
	SecurityToken1	Enable diagnostic mode
Access	SecurityToken2	Enable diagnostic mode
Token	crc(token)==0x18e5a977	Enable developer mode
	crc(token)==0x73bbee22	Enable developer mode
Master Pwd	“3500”	Exit valet mode

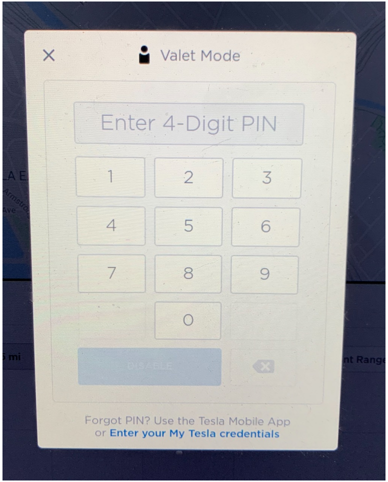
Table: Hidden commands from Tesla firmware.



Application: Egg Hunt in Tesla Infotainment

Category	Content	Description
Easter Egg	"007"	Submarine Easter egg
	"modelxmas"	Show holiday lights
	"42"	Change car name
	"mars"	Turn map into Mars surface
	"transport"	Transport mode
	"performance"	Performance mode
	"showroom"	Showroom mode
Access Token	SecurityToken1	Enable diagnostic mode
	SecurityToken2	Enable diagnostic mode
	crc(token)==0x18e5a977	Enable developer mode
	crc(token)==0x73bbee22	Enable developer mode
Master Pwd	"3500"	Exit valet mode

Table: Hidden commands from Tesla firmware.



Application: Egg Hunt in Tesla Infotainment

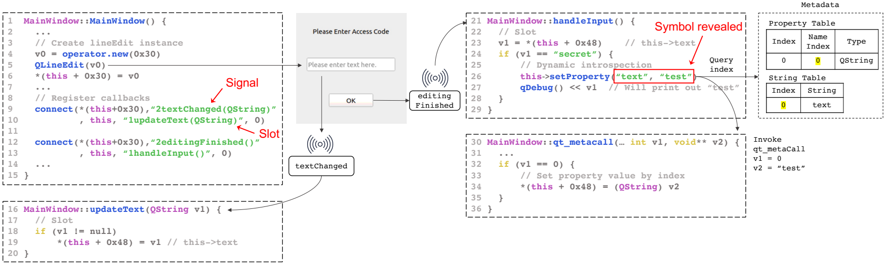
Category	Content	Description
Easter Egg	"007"	Submarine Easter egg
	"modelxmas"	Show holiday lights
	"42"	Change car name
	"mars"	Turn map into Mars surface
	"transport"	Transport mode
	"performance"	Performance mode
	"showroom"	Showroom mode
Access Token	SecurityToken1	Enable diagnostic mode
	SecurityToken2	Enable diagnostic mode
	crc(token)==0x18e5a977	Enable developer mode
	crc(token)==0x73bbee22	Enable developer mode
Master Pwd	"3500"	Exit valet mode

Table: Hidden commands from Tesla firmware.

Disclosure

The Tesla security team acknowledged our findings in 2022/4 and have eliminated the feasible paths for exploiting these hidden commands in the latest firmware.

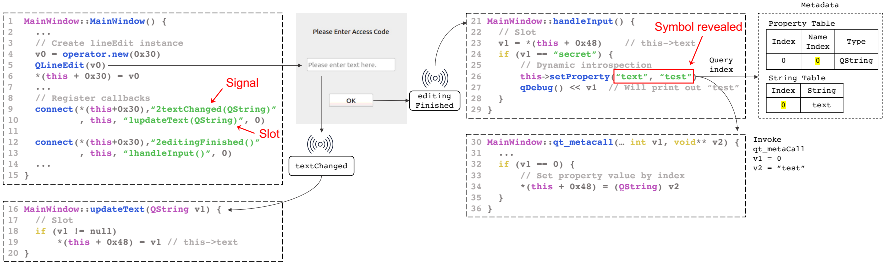
QtRE [USENIX Security'23]



QtRE

- A static analysis tool that leverages Qt's unique insights for function callback and symbol recovery

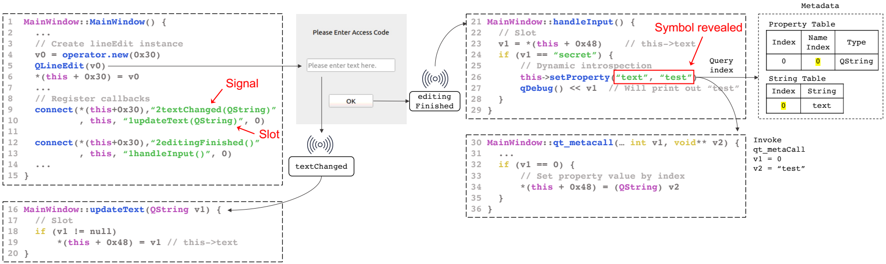
QtRE [USENIX Security'23]



QtRE

- A static analysis tool that leverages Qt's unique insights for function callback and symbol recovery
- It additionally recovered (based on GHIDRA) 10,867 callbacks and 24,973 symbols among 123 binaries

QtRE [USENIX Security'23]



QtRE

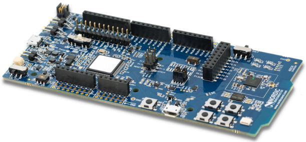
- A static analysis tool that leverages Qt's unique insights for function callback and symbol recovery
- It additionally recovered (based on GHIDRA) 10,867 callbacks and 24,973 symbols among 123 binaries
- We demonstrate an application of input validation analysis with QtRE, and extracted 12 unique hidden commands five new to the public.

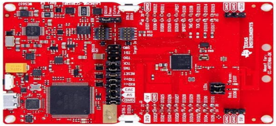
The source code will be released at https://github.com/OSUSecLab/QtRE.

Bluetooth Low Energy

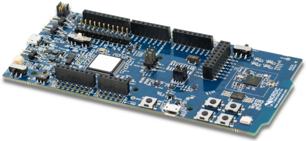
Bluetooth™
4.0
Low Energy

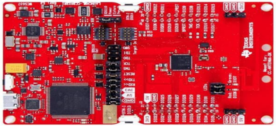
Low Technical Barrier for IoT Development



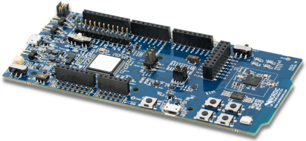


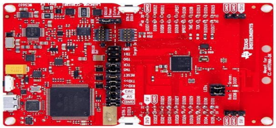
Low Technical Barrier for IoT Development





Low Technical Barrier for IoT Development

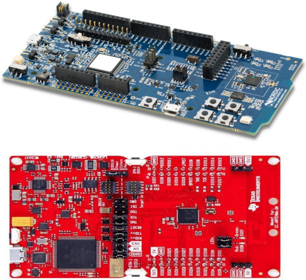




AWS IoT

Azure IoT Hub

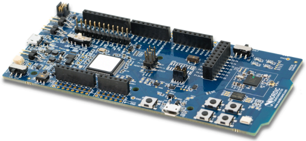
Low Technical Barrier for IoT Development

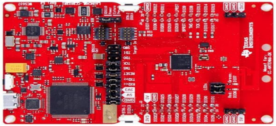


AWS IoT

Azure IoT Hub

Low Technical Barrier for IoT Development



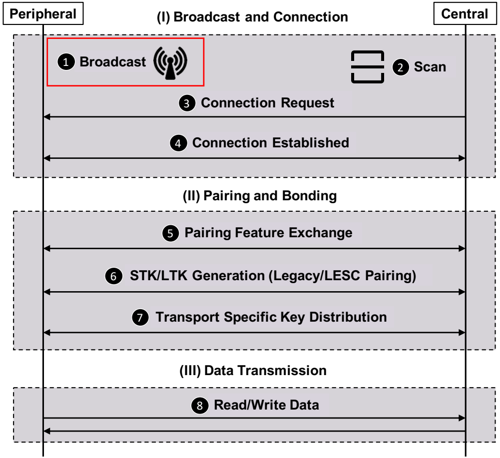


AWS IoT

Azure IoT Hub

Are they secure?

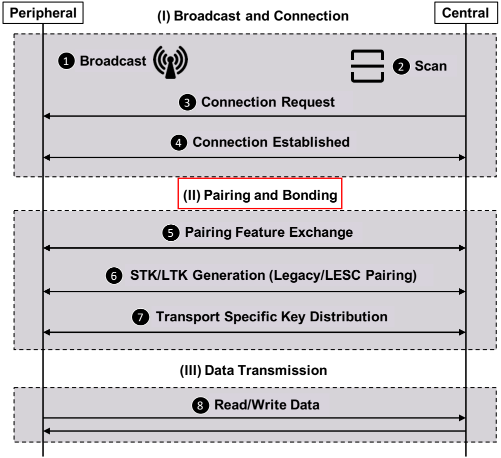
BLE Link Layer Vulnerabilities



Vulnerabilities

1. **Identity Tracking**. Configure static MAC address during broadcast [DPCM16].

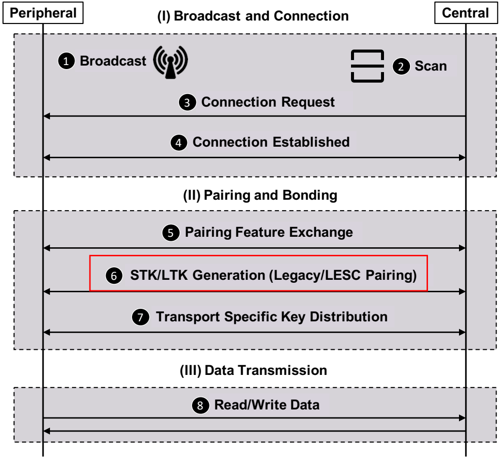
BLE Link Layer Vulnerabilities



Vulnerabilities

1. **Identity Tracking**. Configure static MAC address during broadcast [DPCM16].
2. **Active MITM**. Just Works is adopted as the pairing method.

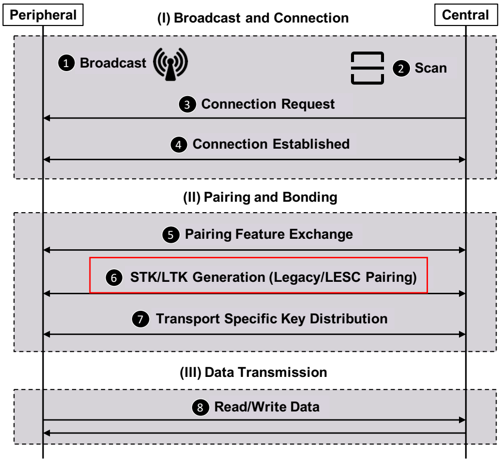
BLE Link Layer Vulnerabilities



Vulnerabilities

1. **Identity Tracking**. Configure static MAC address during broadcast [DPCM16].
2. **Active MITM**. Just Works is adopted as the pairing method.
3. **Passive MITM**. Legacy pairing is used during key exchange [ble14].

BLE Link Layer Vulnerabilities



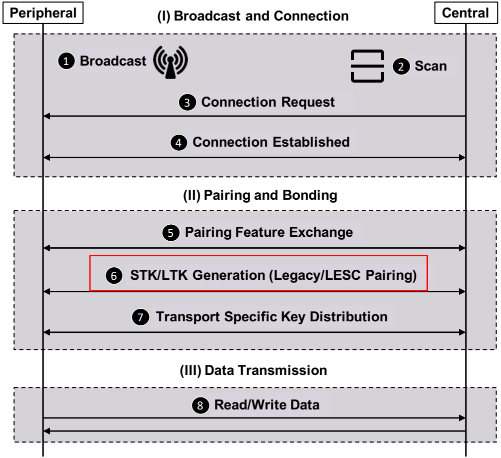
Vulnerabilities

1. **Identity Tracking**. Configure static MAC address during broadcast [DPCM16].
2. **Active MITM**. Just Works is adopted as the pairing method.
3. **Passive MITM**. Legacy pairing is used during key exchange [ble14].

Identification

1. Traffic analysis
2. Mobile app analysis

BLE Link Layer Vulnerabilities



Vulnerabilities

1. **Identity Tracking**. Configure static MAC address during broadcast [DPCM16].
2. **Active MITM**. Just Works is adopted as the pairing method.
3. **Passive MITM**. Legacy pairing is used during key exchange [ble14].

Identification

1. Traffic analysis
2. Mobile app analysis
3. Firmware analysis

An Example of a Just Works Pairing Vulnerability

Read Only Memory

```
243a8   mov   r2, #0x0
243aa   orr   r2, #0x1
243ac   and   r2, #0xe1
243ae   add   r2, #0xc
243b0   and   r2, #0xdf
243b2   ldr   r1, [0x260c8]
243b4   str   r2, [r1,#0x0]
...
25f44   ldr   r2, [0x260c8]
25f46   mov   r1, #0x0
25f48   svc   0x7f
// SD_BLE_GAP_SEC_PARAMS_REPLY
...
260c8   0x20003268
        // ble_gap_sec_parms_t*
```

Register Values

```
r1 = 0x0
r2 = 0x0
```

An Example of a Just Works Pairing Vulnerability

Read Only Memory

```
243a8    mov    r2, #0x0
243aa    orr    r2, #0x1
243ac    and    r2, #0xe1
243ae    add    r2, #0xc
243b0    and    r2, #0xdf
243b2    ldr    r1, [0x260c8]
243b4    str    r2, [r1,#0x0]
...
25f44    ldr    r2, [0x260c8]
25f46    mov    r1, #0x0
25f48    svc    0x7f
// SD_BLE_GAP_SEC_PARAMS_REPLY
...
260c8    0x20003268
         // ble_gap_sec_parms_t*
```

Register Values

```
r1 = 0x0
r2 = 0xD
```

An Example of a Just Works Pairing Vulnerability

Read Only Memory

```
243a8    mov     r2,  #0x0
243aa    orr     r2,  #0x1
243ac    and     r2,  #0xe1
243ae    add     r2,  #0xc
243b0    and     r2,  #0xdf
243b2    ldr     r1,  [0x260c8]
243b4    str     r2,  [r1,#0x0]
...
25f44    ldr     r2,  [0x260c8]
25f46    mov     r1,  #0x0
25f48    svc     0x7f
// SD_BLE_GAP_SEC_PARAMS_REPLY
...
260c8    0x20003268
         // ble_gap_sec_parms_t*
```

Random Access Memory

Struct ble_gap_sec_params_t

```
20003268   uint8  pairing_feature

20003269   uint8  min_key_size
20003270   uint8  max_key_size
20003271   ble_gap_sec_kdist_t  kdist_own
20003275   ble_gap_sec_kdist_t  kdist_peer
```

Register Values

```
r1 = 0x20003268
r2 = 0xD
```

An Example of a Just Works Pairing Vulnerability

Read Only Memory

```
1  243a8   mov   r2, #0x0
2  243aa   orr   r2, #0x1
3  243ac   and   r2, #0xe1
4  243ae   add   r2, #0xc
5  243b0   and   r2, #0xdf
6  243b2   ldr   r1, [0x260c8]
7  243b4   str   r2, [r1,#0x0]
...
8  25f44   ldr   r2, [0x260c8]
9  25f46   mov   r1, #0x0
10 25f48   svc   0x7f
// SD_BLE_GAP_SEC_PARAMS_REPLY
...
11 260c8   0x20003268
           // ble_gap_sec_parms_t*
```

Random Access Memory

Struct ble_gap_sec_params_t

```
20003268   uint8 pairing_feature  = 0xD

20003269   uint8 min_key_size
20003270   uint8 max_key_size
20003271   ble_gap_sec_kdist_t  kdist_own
20003275   ble_gap_sec_kdist_t  kdist_peer
```

Register Values

```
r1 = 0x20003268
r2 = 0xD
```

An Example of a Just Works Pairing Vulnerability

Read Only Memory

```
243a8   mov   r2, #0x0
243aa   orr   r2, #0x1
243ac   and   r2, #0xe1
243ae   add   r2, #0xc
243b0   and   r2, #0xdf
243b2   ldr   r1, [0x260c8]
243b4   str   r2, [r1,#0x0]
...
25f44   ldr   r2, [0x260c8]
25f46   mov   r1, #0x0
25f48   svc   0x7f
// SD_BLE_GAP_SEC_PARAMS_REPLY
...
260c8   0x20003268
        // ble_gap_sec_parms_t*
```

Random Access Memory

Struct ble_gap_sec_params_t

```
20003268   uint8 pairing_feature  = 0xD

20003269   uint8 min_key_size
20003270   uint8 max_key_size
20003271   ble_gap_sec_kdist_t  kdist_own
20003275   ble_gap_sec_kdist_t  kdist_peer
```

Register Values

```
r1 = 0x0
r2 = 0x20003268
```

An Example of a Just Works Pairing Vulnerability

Read Only Memory

```
1  243a8   mov    r2, #0x0
2  243aa   orr    r2, #0x1
3  243ac   and    r2, #0xe1
4  243ae   add    r2, #0xc
5  243b0   and    r2, #0xdf
6  243b2   ldr    r1, [0x260c8]
7  243b4   str    r2, [r1,#0x0]
...
8  25f44   ldr    r2, [0x260c8]
9  25f46   mov    r1, #0x0
10 25f48   svc    0x7f
// SD_BLE_GAP_SEC_PARAMS_REPLY
...
11 260c8   0x20003268
           // ble_gap_sec_parms_t*
```

Random Access Memory

Struct ble_gap_sec_params_t

```
20003268  uint8 pairing_feature  = 0xD
```

BOND	MITM	IO	OOB

```
            // BOND = 1, MITM = 0
            // IO   = 3, OOB  = 0
20003269  uint8 min_key_size
20003270  uint8 max_key_size
20003271  ble_gap_sec_kdist_t  kdist_own
20003275  ble_gap_sec_kdist_t  kdist_peer
```

Register Values

r1 = 0x0
r2 = 0x20003268

An Example of a Just Works Pairing Vulnerability

Correct Firmware Disassembling

Read Only Memory

```
243a8    mov    r2, #0x0
243aa    orr    r2, #0x1
243ac    and    r2, #0xe1
243ae    add    r2, #0xc
243b0    and    r2, #0xdf
243b2    ldr    r1, [0x260c8]
243b4    str    r2, [r1,#0x0]
...
25f44    ldr    r2, [0x260c8]
25f46    mov    r1, #0x0
25f48    svc    0x7f
// SD_BLE_GAP_SEC_PARAMS_REPLY
...
260c8    0x20003268
         // ble_gap_sec_parms_t*
```

Random Access Memory

Struct ble_gap_sec_params_t

```
20003268   uint8 pairing_feature  = 0xD
```

BOND	MITM	IO	OOB

```
             // BOND  = 1, MITM  = 0
             // IO    = 3, OOB   = 0
20003269   uint8 min_key_size
20003270   uint8 max_key_size
20003271   ble_gap_sec_kdist_t  kdist_own
20003275   ble_gap_sec_kdist_t  kdist_peer
```

Register Values

r1 = 0x0

r2 = 0x20003268

An Example of a Just Works Pairing Vulnerability

Correct Firmware Disassembling

Read Only Memory

```
243a8   mov   r2, #0x0
243aa   orr   r2, #0x1
243ac   and   r2, #0xe1
243ae   add   r2, #0xc
243b0   and   r2, #0xdf
243b2   ldr   r1, [0x260c8]
243b4   str   r2, [r1,#0x0]
...
25f44   ldr   r2, [0x260c8]
25f46   mov   r1, #0x0
25f48   svc   0x7f
// SD_BLE_GAP_SEC_PARAMS_REPLY
...
260c8         0x20003268
              // ble_gap_sec_parms_t*
```

Recognize data structures

Random Access Memory

Struct ble_gap_sec_params_t

```
20003268   uint8 pairing_feature  = 0xD
```

BOND	MITM	IO	OOB

```
           // BOND  = 1, MITM  = 0
           // IO    = 3, OOB   = 0
20003269   uint8 min_key_size
20003270   uint8 max_key_size
20003271   ble_gap_sec_kdist_t  kdist_own
20003275   ble_gap_sec_kdist_t  kdist_peer
```

Register Values

```
r1 = 0x0
r2 = 0x20003268
```

An Example of a Just Works Pairing Vulnerability

Correct Firmware Disassembling → **Read Only Memory**

```
243a8  mov  r2, #0x0
243aa  orr  r2, #0x1
243ac  and  r2, #0xe1
243ae  add  r2, #0xc
243b0  and  r2, #0xdf
243b2  ldr  r1, [0x260c8]
243b4  str  r2, [r1,#0x0]
...
25f44  ldr  r2, [0x260c8]
25f46  mov  r1, #0x0
25f48  svc  0x7f
// SD_BLE_GAP_SEC_PARAMS_REPLY
...
260c8       0x20003268
            // ble_gap_sec_parms_t*
```

Recognize data structures → **Random Access Memory**

Struct ble_gap_sec_params_t

```
20003268  uint8 pairing_feature  = 0xD
```

BOND	MITM	IO	OOB

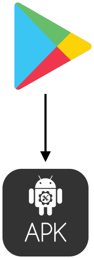
```
          // BOND  = 1, MITM  = 0
          // IO    = 3, OOB   = 0
20003269  uint8 min_key_size
20003270  uint8 max_key_size
20003271  ble_gap_sec_kdist_t  kdist_own
20003275  ble_gap_sec_kdist_t  kdist_peer
```

Value computation → **Register Values**

```
r1 = 0x0
r2 = 0x20003268
```

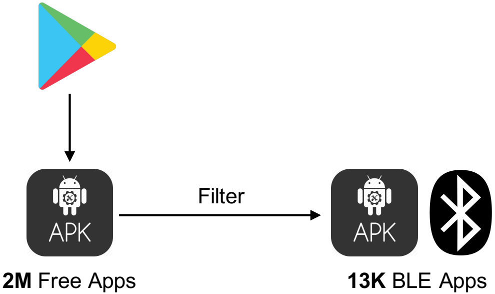
Firmware Collection

Firmware Collection

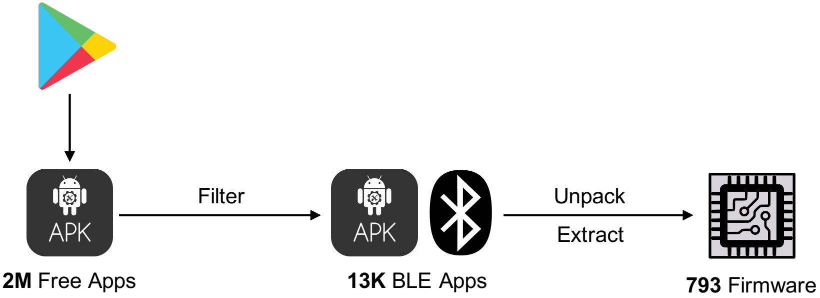


2M Free Apps

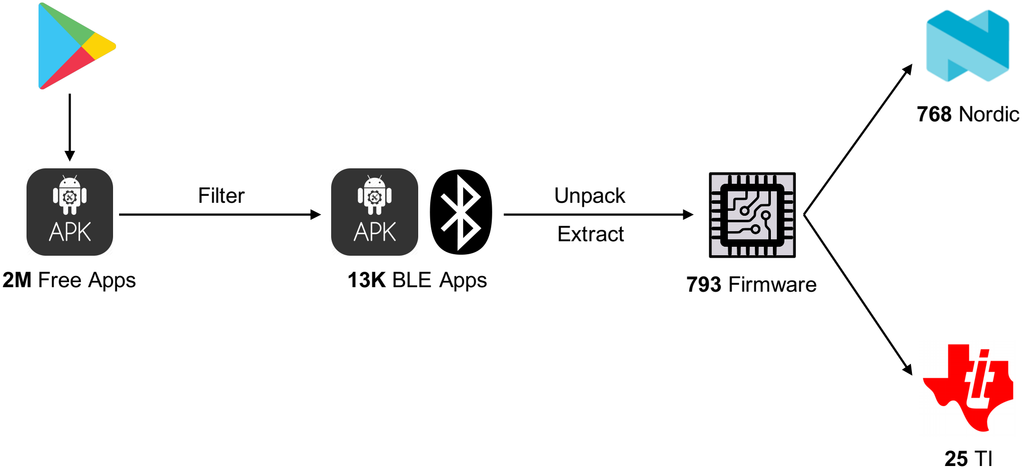
Firmware Collection



Firmware Collection



Firmware Collection



Experiment Results

Identity Tracking Vulnerability Identification

Among the 538 devices, nearly all of them (**98.1%**) have configured random static addresses that do not change periodically.

Experiment Results

Identity Tracking Vulnerability Identification

Among the 538 devices, nearly all of them (**98.1%**) have configured random static addresses that do not change periodically.

Firmware Name	Mobile App	Category	# Device
cogobeacon	com.aegismobility.guardian	Car Accessory	4
sd_bl	fr.solem.solemwf	Agricultural Equip.	2
LRFL_nRF52	fr.solem.solemwf	Agricultural Equip.	2
orb	one.shade.app	Smart Light	1
sd_bl	com.rainbird	Agricultural Equip.	1

Table: Firmware using private MAC address.

Experiment Results

Active MITM Vulnerability Identification

385 (71.5%) devices use Just Works pairing, which essentially does not provide any protection against active MITM attacks at the BLE link layer.

Experiment Results

Active MITM Vulnerability Identification

385 (71.5%) devices use Just Works pairing, which essentially does not provide any protection against active MITM attacks at the BLE link layer.

Item	N	T	Total	%
# Total Device	513	25	538	100
# Device w/ active MITM vulnerability	384	1	385	71.5
# Device w/ Just Works pairing only	317	1	318	59.1
# Device w/ flawed Passkey implementation	37	0	37	6.9
# Device w/ flawed OOB implementation	30	0	30	5.6
# Device w/ secure pairing	6	24	30	3.8
# Device w/ correct Passkey implementation	3	24	27	3.4
# Device w/ correct OOB implementation	3	0	3	0.4

Table: Pairing configurations of devices (N:Nordic, T:TI).

Experiment Results

Passive MITM Vulnerability Identification

98.5% of the devices fail to enforce LESC pairing, and thus they can be vulnerable to passive MITM attacks if there is no application-layer encryption.

Experiment Results

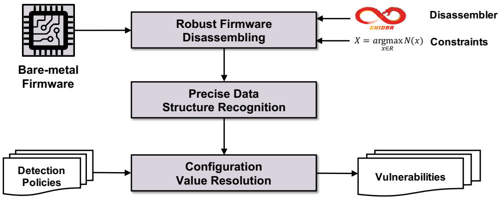
Passive MITM Vulnerability Identification

98.5% of the devices fail to enforce LESC pairing, and thus they can be vulnerable to passive MITM attacks if there is no application-layer encryption.

Firmware Name	Mobile App	Category	# Version
DogBodyBoard	com.wowwee.chip	Robot	16
BW_Pro	com.ecomm.smart_panel	Tag	1
Smart_Handle	com.exitec.smartlock	Smart Lock	1
Sma05	com.smalife.watch	Wearable	1
CPRmeter	com.laerdal.cprmeter2	Medical Device	4
WiJumpLE	com.wesssrl.wijumple	Sensor	1
nRF Beacon	no.nordicsemi.android.nrfbeacon	Beacon	1
Hoot Bank	com.qvivr.hoot	Debit Card	1

Table: Firmware that enforce LESC pairing.

FirmXRay [CCS'20]



FirmXRay

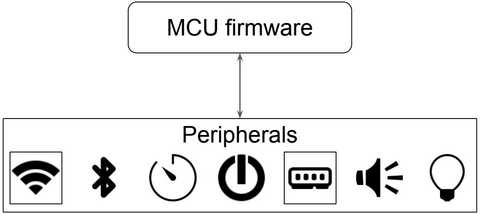
- A static analysis tool based on Ghidra for detecting BLE link layer vulnerabilities from bare-metal firmware.
- A scalable approach to efficiently collect bare-metal firmware images from only mobile apps.
- Vulnerability discovery and attack case studies.

The source code is available at https://github.com/OSUSecLab/FirmXRay.

Microcontroller Unit (MCU)

- The chip inside the board
- Ubiquitous (e.g., drone, smart light bulb)

Microcontroller Unit (MCU)



- Peripherals are inside the provided board
- Firmware controls peripherals through peripheral registers
- Peripheral executes firmware through the corresponding interrupt

Microcontroller Unit (MCU)

MCU Firmware Vulnerabilities

1. Memory corruption
2. Privacy leakage
3. Peripheral malfunctioning

Microcontroller Unit (MCU)

MCU Firmware Vulnerabilities

1. Memory corruption
2. Privacy leakage
3. Peripheral malfunctioning

Firmware Analysis

1. **Hardware-in-the-loop**. Testing firmware with hardware
2. **Re-hosting**. Emulating firmware without hardware

Microcontroller Unit (MCU)

MCU Firmware Vulnerabilities

1. Memory corruption
2. Privacy leakage
3. Peripheral malfunctioning

Firmware Analysis

1. **Hardware-in-the-loop**. Testing firmware with hardware
2. **Re-hosting**. Emulating firmware without hardware

Common Challenge

Modeling Peripheral Processing

An Example of Processing a Peripheral Register

Execution just based on the firmware code

```
REG_CLOCK = 0x40023800;
*REG_CLOCK = 0x1000000; // set 24-bit
if (*REG_CLOCK & 0x2000000) == 0) { // check 25-bit
 return HAL_ERROR;
}
Freq = HAL_RCC_GetSysClockFreq();
return HAL_OK;
```

```
[REG_CLOCK] 0x40023800 = <uninitialized>
```

An Example of Processing a Peripheral Register

Execution just based on the firmware code

```
1:  REG_CLOCK = 0x40023800;
2:  *REG_CLOCK = 0x1000000; // set 24-bit
3:  if (*REG_CLOCK & 0x2000000) == 0) { // check 25-bit
4:   return HAL_ERROR;
5:  }
6:  Freq = HAL_RCC_GetSysClockFreq();
7:  return HAL_OK;
```

```
[REG_CLOCK] 0x40023800 = 0x1000000
```

An Example of Processing a Peripheral Register

Execution just based on the firmware code

```
1:  REG_CLOCK = 0x40023800;
2:  *REG_CLOCK = 0x1000000; // set 24-bit
3:  if (*REG_CLOCK & 0x2000000) == 0) { // check 25-bit
4:   return HAL_ERROR;
5:  }
6:  Freq = HAL_RCC_GetSysClockFreq();
7:  return HAL_OK;
```

```
[REG_CLOCK] 0x40023800 = 0x1000000
```

An Example of Processing a Peripheral Register

Execution just based on the <u>firmware code</u>

```
REG_CLOCK = 0x40023800;
*REG_CLOCK = 0x1000000; // set 24-bit
if (*REG_CLOCK & 0x2000000) == 0) { // check 25-bit
 return HAL_ERROR;
}
Freq = HAL_RCC_GetSysClockFreq();
return HAL_OK;
```

```
[REG_CLOCK] 0x40023800 = 0x1000000
```

An Example of Processing a Peripheral Register

Execution on real MCU hardware

```
1:  REG_CLOCK = 0x40023800;
2:  *REG_CLOCK = 0x1000000; // set 24-bit
3:  if (*REG_CLOCK & 0x2000000) == 0) { // check 25-bit
4:   return HAL_ERROR;
5:  }
6:  Freq = HAL_RCC_GetSysClockFreq();
7:  return HAL_OK;
```

```
[REG_CLOCK] 0x40023800 = 0x3000000
```

An Example of Processing a Peripheral Register

Execution on real MCU hardware

```
REG_CLOCK = 0x40023800;
*REG_CLOCK = 0x1000000; // set 24-bit
if (*REG_CLOCK & 0x2000000) == 0) { // check 25-bit
 return HAL_ERROR;
}
Freq = HAL_RCC_GetSysClockFreq();
return HAL_OK;
```

```
[REG_CLOCK] 0x40023800 = 0x3000000
```

An Example of Processing a Peripheral Register

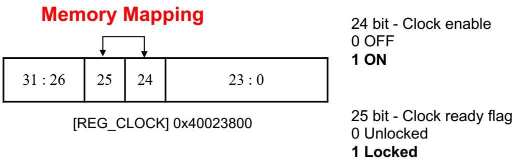
Execution on real MCU hardware

```
REG_CLOCK = 0x40023800;
*REG_CLOCK = 0x1000000; // set 24-bit
if (*REG_CLOCK & 0x2000000) == 0) { // check 25-bit
 return HAL_ERROR;
}
Freq = HAL_RCC_GetSysClockFreq();
return HAL_OK;
```

```
[REG_CLOCK] 0x40023800 = 0x3000000
```

Hidden Memory Mapping

Peripheral register bits get simultaneously updated by the MCU hardware
As some bits are semantically relevant (e.g., clock status)

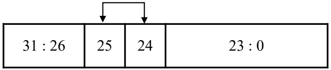


Hidden Memory Mapping

Peripheral register bits get simultaneously updated by the MCU hardware
As some bits are semantically relevant (e.g., clock status)

```
REG_CLOCK = 0x40023800;
*REG_CLOCK = 0x1000000; // set 24-bit
if (*REG_CLOCK & 0x2000000) == 0) {
 return HAL_ERROR;
}
Freq = HAL_RCC_GetSysClockFreq();
return HAL_OK;
```

Memory Mapping



[REG_CLOCK] 0x40023800

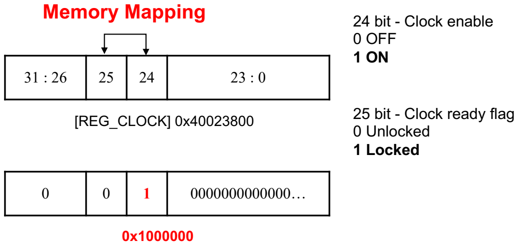
24 bit - Clock enable
0 OFF
1 ON

25 bit - Clock ready flag
0 Unlocked
1 Locked

Hidden Memory Mapping

Peripheral register bits get simultaneously updated by the MCU hardware
As some bits are semantically relevant (e.g., clock status)

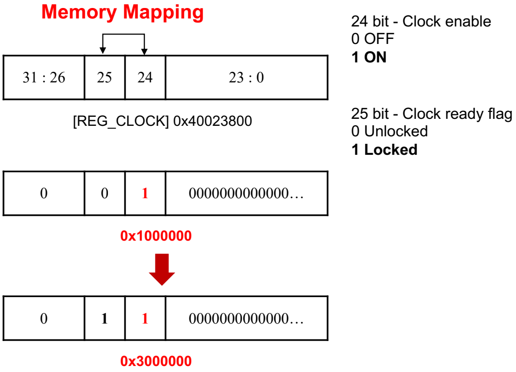
```
REG_CLOCK = 0x40023800;
*REG_CLOCK = 0x1000000; // set 24-bit
if (*REG_CLOCK & 0x2000000) == 0) {
 return HAL_ERROR;
}
Freq = HAL_RCC_GetSysClockFreq();
return HAL_OK;
```



Hidden Memory Mapping

Peripheral register bits get simultaneously updated by the MCU hardware
As some bits are semantically relevant (e.g., clock status)

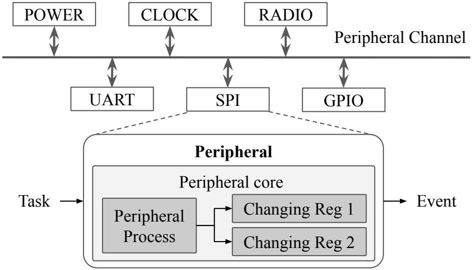
```
REG_CLOCK = 0x40023800;
*REG_CLOCK = 0x1000000; // set 24-bit
if (*REG_CLOCK & 0x2000000) == 0) {
 return HAL_ERROR;
}
Freq = HAL_RCC_GetSysClockFreq();
return HAL_OK;
```



Hidden Memory Mapping

Root cause: Autonomous Peripheral Operation

Hardware feature in microcontroller architectures. The peripheral performs its operation without CPU intervention to save energy.



Hidden Memory Mapping

Root cause: Autonomous Peripheral Operation

Hardware feature in microcontroller architectures. The peripheral performs its operation without CPU intervention to save energy.

Bit 24 PLLRDY: Main PLL (PLL) clock ready flag

Set by hardware to indicate that PLL is locked.
0: PLL unlocked
1: PLL locked

Bit 1 SBF: Standby flag

This bit is set by hardware and cleared only by a POR/PDR (power-on reset/power-down reset) or by setting the CSBF bit in the PWR_CR register
0: Device has not been in Standby mode
1: Device has been in Standby mode

AutoMap Overview

Challenges

1. Nearly infinite number of possible writes to peripheral registers
2. Cannot infer memory mappings from code-level
3. Dependency of peripheral register writes

AutoMap Overview

Challenges

1. Nearly infinite number of possible writes to peripheral registers
2. Cannot infer memory mappings from code-level
3. Dependency of peripheral register writes

Solutions

1. On-demand memory mapping inference
2. Differential memory introspection through hardware-in-the-loop
3. Memory context preparation by executing previous peripheral registers write intrusions

Experiment Setup

- Three MCUs
  - Nordic NRF52832
    - 41 example firmware included in SDK
  - STMicroelectronics STM32F103
    - 5 real-world firmware from μEMU [ZGLZ21]
  - STMicroelectronics STM32F429
    - 4 real-world firmware from μEMU [ZGLZ21]

Experiment Results

Identity Memory Mapping in Example Firmware

At least one memory mapping is discovered in every firmware. Even single register write can affect multiple other registers.

Experiment Results

Identity Memory Mapping in Example Firmware

At least one memory mapping is discovered in every firmware. Even single register write can affect multiple other registers.

MCU	Firmware	# of Writes Causing M.M	Max # of M.M by single write
NRF52832	bk_freertos	21	7
	bk	9	3
	bk_rtc	21	7
	bk_systick	9	3
	bsp	35	11

Table: Memory mapping result on example firmware of NRF52832

Experiment Results

Integrating AutoMap with μEMU

AutoMap with μEMU can cover at most 15.59% more basic blocks than μEMU.

Experiment Results

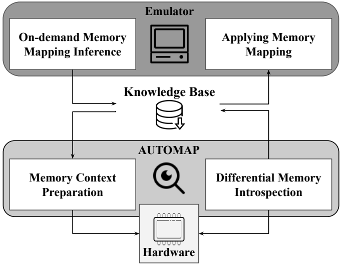
Integrating AutoMap with μEMU

AutoMap with μEMU can cover at most 15.59% more basic blocks than μEMU.

Firmware	# executed BBs		BBs portion of AutoMap not in μEMU	
	AutoMap	μEMU	#	%
Drone	1,413	1,410	5	0.35%
Gateway	1,385	1,248	216	15.59%
Steering_Iron	1,402	1,289	116	8.27%
Reflow_Oven	845	830	17	2.01%
Robot	1,035	964	77	7.43%

Table: Fuzzing result comparison between μEMU and both AutoMap and μEMU.

AutoMap [RAID'22]



AutoMap

- Discover memory mapping in peripheral registers.
- Propose AutoMap to discover memory mappings systematically.
- Emulate firmware properly with memory mappings and execute more basic blocks when AutoMap integrates with μEMU.

The source code is available at https://github.com/OSUSecLab/AutoMap.

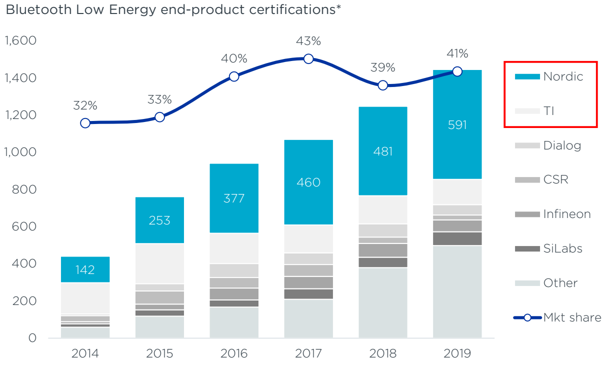
Takeaways

- The need to analyze new domains for heterogeneous IoT binary analysis
- New domains (mechanisms, architecture, API...) lead to new *insights* and *techniques*

The Potentials of Domain-Aware Analysis

Name	Category	# Repository	%
Qt	Framework	45,635	35.70%
ROS	Robotics	16,796	13.14%
Boost	Framework	6,205	4.85%
MFC	Framework	4,409	3.45%
Cocos2d	Game Engine	3,587	2.81%
OpenFrameworks	Framework	3,264	2.55%
JUCE	Framework	2,204	1.72%
PCL	Robotics	1,719	1.34%
imgui	GUI	1,557	1.22%
wxWidgets	GUI	1,076	0.84%
Cinder	Framework	1,042	0.82%
Allegro	Game Engine	958	0.75%
Godot	Game Engine	682	0.53%
GamePlay	Game Engine	561	0.44%
dlib	Framework	547	0.43%
FLTK	GUI	518	0.41%
GTK++	GUI	436	0.34%
LibU	Framework	425	0.33%
raylib	Game Engine	376	0.29%
gtkmm	GUI	349	0.27%

Top C++ frameworks for software development.



Data Source: Nordic Quarterly Presentation Q4 2019

The Potentials of Domain-Aware Analysis

1. Systematically vetting domain-specific applications

The Potentials of Domain-Aware Analysis

1. Systematically vetting domain-specific applications
2. Extension to other IoT domains, architectures, frameworks...

The Potentials of Domain-Aware Analysis

1. Systematically vetting domain-specific applications
2. Extension to other IoT domains, architectures, frameworks...
3. Support various security applications (e.g., Qt-Fuzz, Automap-Fuzz)

The Potentials of Domain-Aware Analysis

1. Systematically vetting domain-specific applications
2. Extension to other IoT domains, architectures, frameworks...
3. Support various security applications (e.g., Qt-Fuzz, Automap-Fuzz)
4. Generalize methodology and insights to other similar domains

Thank You

Unlocking the Potential of Domain Aware Binary Analysis in the Era of IoT

Zhiqiang Lin

zlin@cse.ohio-state.edu

April 18th, 2023

References I

Bluetooth specification version 4.2, https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=286439, 2014.

Aveek K Das, Parth H Pathak, Chen-Nee Chuah, and Prasant Mohapatra, *Uncovering privacy leakage in ble network traffic of wearable fitness trackers*, Proceedings of the 17th international workshop on mobile computing systems and applications, 2016, pp. 99–104.

Ghidra, https://ghidra-sre.org/.

Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida, *Marx: Uncovering class hierarchies in c++ programs.*, NDSS, 2017.

Edward J Schwartz, Cory F Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S Havrilla, and Charles Hines, *Using logic programming to recover c++ classes and methods from compiled executables*, Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 426–441.

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al., *Sok:(state of) the art of war: Offensive techniques in binary analysis*, 2016 IEEE Symposium on Security and Privacy (SP), IEEE, 2016, pp. 138–157.

Hieu Tran, Ngoc Tran, Son Nguyen, Hoan Nguyen, and Tien N Nguyen, *Recovering variable names for minified code with usage contexts*, 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 1165–1175.

Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida, *A tough call: Mitigating advanced code-reuse attacks at the binary level*, 2016 IEEE Symposium on Security and Privacy (SP), IEEE, 2016, pp. 934–953.

Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang, *Automatic firmware emulation through invalidity-guided knowledge inference.*, USENIX Security Symposium, 2021, pp. 2007–2024.