BaseComp: A Comparative Analysis for Integrity Protection in Cellular Baseband Software

Eunsoo Kim*, Min Woo Baek*, CheolJun Park†, Dongkwan Kim‡, Yongdae Kim†, Insu Yun†

†KAIST, ‡Samsung SDS
Baseband Software
Cellular Network Architecture
Baseband Software
Attack Scenario

Application Processor
Baseband Processor

Realtime Software

Malicious Base Station

LTE
Baseband Software
Message Processing Logic

Layer 3 Protocol Message
⇒ Security Checker
⇒ Message Decoder
⇒ Message Dispatcher
⇒ Handler A
⇒ Handler B
Baseband Software

Challenges

- Obscurity
  - Vendors don't release the details

- Large Binary Size
  - The baseband software has to implement documents of n*100 pages
Motivation

Existing Approaches

• Dynamic Analysis
  • DoLTEst (Security'22), Firmwire (NDSS'22)
  • Sends messages and observes responses from real or emulated devices
  • Has to restrict the search space leading to missing bugs

• Static Analysis / BaseSpec (NDSS'21)
  • Limited to message decoding and fails to analyze integrity protection
  • The vast size and obscurity causes highly resource-consuming manual analysis
Motivation

Our Approach

• Static Analysis
  • Without having to restrict the search space

• Comparative Analysis
  • Comparison with specification to uncover bugs in integrity protection

• Probabilistic Inference
  • Reduce the amount of manual effort needed
BaseComp
Overview

Firmware

Probabilistic Inference

Manual Analysis

Symbolic Analysis

Implementation Model

Reference Model

Comparative Analysis

Mismatches

Specification

Specification Analysis
BaseComp
Probabilistic Inference

Find the integrity protection function.
The integrity protection function needs to have the following logics.

- Encryption/decryption using AES/ZUC/SNOW3G
- Message type filtering based on subclause 4.4.4.2 of TS 24.301
The integrity protection function needs to have the following logics.

- Encryption/decryption using AES/ZUC/SNOW3G
- Message type filtering based on subclause 4.4.4.2 of TS 24.301

Steps
1. Identifying MAC functions.
2. Identifying message type comparing functions.
3. Putting it all together.
BaseComp
Probabilistic Inference

1. Identifying MAC functions.

- Cryptographic functions identified by magic constants (S-Box)
1. Identifying MAC functions.

• Find common ancestors of cryptographic functions
1. Identifying MAC functions.

- Prioritize lower common ancestors

<Call Graph>
2. Identifying message type comparing functions.

4.4.4.2 Integrity checking of NAS signalling messages in the UE

Except the messages listed below, no NAS signalling messages shall be processed by the receiving EMM entity in the UE or forwarded to the ESM entity, unless the network has established secure exchange of NAS messages for the NAS signalling connection:

- EMM messages:
  - IDENTITY REQUEST (if requested identification parameter is IMSI);
  - AUTHENTICATION REQUEST;
  - AUTHENTICATION REJECT;
  - ATTACH REJECT (if the EMM cause is not #25);
  - DETACH ACCEPT (for non switch off);
  - TRACKING AREA UPDATE REJECT (if the EMM cause is not #25);
  - SERVICE REJECT (if the EMM cause is not #25).

NOTE: These messages are accepted by the UE without integrity protection, as in certain situations they are sent by the network before security can be activated.
BaseComp
Probabilistic Inference

2. Identifying message type comparing functions.

4.4.4.2 Integrity checking of NAS signalling messages in the UE

Except the messages listed below, no NAS signalling messages shall be processed by the receiving EMM entity in the UE or forwarded to the ESM entity, unless the network has established secure exchange of NAS messages for the NAS signalling connection:

- EMM messages:
  - IDENTITY REQUEST (if requested identification parameter is IMSI);
  - AUTHENTICATION REQUEST;
  - AUTHENTICATION REJECT;
  - ATTACH REJECT (if the EMM cause is not #25);
  - DETACH ACCEPT (for non switch off);
  - TRACKING AREA UPDATE REJECT (if the EMM cause is not #25);
  - SERVICE REJECT (if the EMM cause is not #25).

NOTE: These messages are accepted by the UE without integrity protection, as in certain situations by the network before security can be activated.

• Iterate every function and its variables
3. Putting it all together.

- Prioritize lower common ancestors
- Find common ancestors of
  - MAC function
  - Message type comparing function
BaseComp

Manual Analysis

Gather additional information to do symbolic execution.

- Firmware
- Probabilistic Inference
- Manual Analysis
- Specification Analysis
- Symbolic Analysis
- Implementation Model
- Reference Model
- Comparative Analysis
- Mismatches
BaseComp
Manual Analysis

• Additional information about the firmware is required to process symbolic execution

```
def symbolize(s, config):
    # Symbolizes a message buffer and a state variable
    msg_buf = s.solver.BVS('message_buffer', 32)
    s.regs.r0 = msg_buf

    sec_state = s.solver.BVS('security_state', 8)
    s.memory.store(config.security_state, sec_state)

def accepting(s, config):
    # Check if this return represents accepting a message
    return s.ret_val == 1
```

• Vendor-specific analysis module
  • How to symbolize variables
  • How to decide if a message is accepted

• Required per-vendor
BaseComp
Manual Analysis

- Additional information about the firmware is required to process symbolic execution

```python
def symbolize(s, config):
    # Symbolizes a message buffer and a state variable
    msg_buf = s.solver.BVS('message_buffer', 32)
    s.regs.r0 = msg_buf
    sec_state = s.solver.BVS('security_state', 8)
    s.memory.store(config.security_state, sec_state)

def accepting(s, config):
    # Check if this return represents accepting a message
    return s.ret_val == 1
```

- Vendor-specific analysis module
  - How to symbolize variables
  - How to decide if a message is accepted
- Required per-vendor
BaseComp

Manual Analysis

- Additional information about the firmware is required to process symbolic execution

```python
def symbolize(s, config):
    # Symbolizes a message buffer and a state variable
    msg_buf = s.solver.BVS('message_buffer', 32)
    s.regs.r0 = msg_buf

    sec_state = s.solver.BVS('security_state', 8)
    s.memory.store(config.security_state, sec_state)

def accepting(s, config):
    # Check if this return represents accepting a message
    return s.ret_val == 1
```

- Vendor-specific analysis module
  - How to symbolize variables
  - How to decide if a message is accepted
  - Required per-vendor
BaseComp
Manual Analysis

• Additional information about the firmware is required to process symbolic execution

```python
1 def symbolize(s, config):
2     # Symbolizes a message buffer and a state variable
3     msg_buf = ssolver.BVS('message_buffer', 32)
4     sregs.r0 = msg_buf
5     
6     sec_state = ssolver.BVS('security_state', 8)
7     s.memory.store(config.security_state, sec_state)

8 def accepting(s, config):
9     # Check if this return represents accepting a message
10    return s.ret_val == 1
```

• Vendor-specific analysis module
  • How to symbolize variables
  • How to decide if a message is accepted

• Required per-vendor

• Firmware-specific configuration
  • Integrity protection function address
  • MAC validation function address
  • Security state address
  • Deny-list of functions to prevent path explosion

• Required per-firmware
BaseComp
Manual Analysis

• Additional information about the firmware is required to process symbolic execution

```python
1 def symbolize(s, config):
2    # Symbolizes a message buffer and a state variable
3    msg_buf = s.solver.BVS('message_buffer', 32)
4    s.regs.r0 = msg_buf
5
6    sec_state = s.solver.BVS('security_state', 8)
7    s.memory.store(config.security_state, sec_state)
8
9 def accepting(s, config):
10    # Check if this return represents accepting a message
11    return s.ret_val == 1
```

• Vendor-specific analysis module
  • How to symbolize variables
  • How to decide if a message is accepted
• Required per-vendor

• Firmware-specific configuration
  • Integrity protection function address
  • MAC validation function address
  • Security state address
  • Deny-list of functions to prevent path explosion
• Required per-firmware

analysis: ./analysis_samsung.py

# Functions for analysis
integrity_func: 0x4150AEC
mac_validation_func: 0x4150A3D6
security_state: 0x429B27C4

# Functions to skip to avoid path explosion
skip_funcs:
- 0x40CECC87
- 0x4057F5FB
BaseComp
Manual Analysis

- Additional information about the firmware is required to process symbolic execution

```
def symbolize(s, config):
    # Symbolizes a message buffer and a state variable
    msg_buf = s.solver.BVS('message_buffer', 32)
    s.regs.r0 = msg_buf

    sec_state = s.solver.BVS('security_state', 8)
    s.memory.store(config.security_state, sec_state)

def accepting(s, config):
    # Check if this return represents accepting a message
    return s.ret_val == 1
```

- Vendor-specific analysis module
  - How to symbolize variables
  - How to decide if a message is accepted
  - Required per-vendor

- Firmware-specific configuration
  - Integrity protection function address
  - MAC validation function address
  - Security state address
  - Deny-list of functions to prevent path explosion
  - Required per-firmware
Collect constraints from the integrity protection function.
BaseComp
Symbolic Analysis

- Under-constrained symbolic execution on the integrity protection function.
- Collect constraints related to the message.
BaseComp
Symbolic Analysis

- Under-constrained symbolic execution on the integrity protection function.
- Collect constraints related to the message.

```c
// A state variable for a security context.
SecState sec_state;

bool IntegrityProtection(void* message) {
    // Returns true if the 'message' is valid to be accepted.
    if (CheckHeader(message)
        && (!IsProtected(message) || CheckSeq(message))
        && (!IsProtected(message) || ValidateMac(message))) return true;
    else
        return false;
}

bool CheckHeader(void* message) {
    uint8_t sec_hdr_type = GetSecHdrType(message);
    uint8_t msg_type = GetMsgType(message);

    if (sec_state == SECURE) {
        if (sec_hdr_type == 0)
            return false;
        else if (sec_hdr_type != 0 && sec_hdr_type == 3)
            return true;
        else
            return false;
    } else { // INSECURE
        if (sec_hdr_type == 0) {
            switch (msg_type) {
                case 0x55:
                case 0x44:
                case 0x48:
                case 0x4E:
                case 0x52:
                case 0x54:
                case 0x46:
                    return true;
                default:
                    return false;
            }
        }
    }
}
BaseComp
Symbolic Analysis

- Under-constrained symbolic execution on the integrity protection function.
- Collect constraints related to the message.

```
sec_state == SECURE +
0 < sec_hdr_type <= 3
```
BaseComp
Symbolic Analysis

• Under-constrained symbolic execution on the integrity protection function.

• Collect constraints related to the message.

```cpp
bool IntegrityProtection(void* message) {
    // A state variable for a security context.
    SecState sec_state;

    if (CheckHeader(message) && (!IsProtected(message) || CheckSeq(message))
        && (!IsProtected(message) || ValidateMac(message))) return true;
    else
        return false;
}

bool CheckHeader(void* message) {
    uint8_t sec_hdr_type = GetSecHdrType(message);
    uint8_t msg_type = GetMsgType(message);

    if (sec_state == SECURE) {
        if (sec_hdr_type == 0)
            return false;
        else if (sec_hdr_type != 0 && sec_hdr_type <= 3)
            return true;
        else
            return false;
    }

    if (sec_state != SECURE)
        return false;

    switch (msg_type) {
        case 0x55:
        case 0x44:
        case 0x4B:
        case 0x4E:
        case 0x52:
        case 0x54:
        case 0x46:
            return true;
        default:
            return false;
    }
}
```
BaseComp
Symbolic Analysis

• Under-constrained symbolic execution on the integrity protection function.

• Collect constraints related to the message.

\[
\begin{align*}
\text{sec\_state} &\equiv \text{SECURE} \\
0 < \text{sec\_hdr\_type} &\leq 3 \\
\text{CheckSeq}(\text{message}) &\equiv \text{true} \land \text{ValidateMac}(\text{message}) \equiv \text{true} \\
\end{align*}
\]

Message is accepted!

```c
// A state variable for a security context.
SecState sec_state;

bool IntegrityProtection(void* message) {
    // Returns true if the 'message' is valid to be accepted.
    if (CheckHeader(message) 
        && ((IsProtected(message) || CheckSeq(message)) 
             && ((IsProtected(message) || ValidateMac(message))))
        return true;
    else
        return false;
}

bool CheckHeader(void* message) {
    uint8_t sec_hdr_type = GetSecHdrType(message);
    uint8_t msg_type = GetMsgType(message);

    if (sec_state == SECURE) {
        if (sec_hdr_type == 0)
            return false;
        else if (sec_hdr_type != 0 && sec_hdr_type <= 3)
            return true;
    } else { // INSECURE
        if (sec_hdr_type == 0) {
            switch (msg_type) {
                case 0x55:
                case 0x44:
                case 0x48:
                case 0x4E:
                case 0x52:
                case 0x54:
                case 0x46:
                    return true;
                default:
                    return false;
            }
        }
    }
}
```
BaseComp
Comparative Analysis

Compare the two models and find mismatches.

Firmware
Speciﬁcation

Probabilistic Inference
Manual Analysis
Symbolic Analysis

Implementation Model
Reference Model
Comparative Analysis

Mismatches
Evaluation

Setup

• Research Questions
  1. How well can BaseComp find the integrity protection function?
  2. How effectively can BaseComp discover bugs?

• Dataset
  • 16 images (10, 5, 1 from Samsung, MediaTek, srsRAN respectively)
  • ARM, MIPS(with 16e2 extension), and x86 architecture
Evaluation

How well can BaseComp find the integrity protection function?

- Effectiveness

<table>
<thead>
<tr>
<th></th>
<th>G950</th>
<th>G955</th>
<th>G960</th>
<th>G965</th>
<th>G970</th>
<th>G975</th>
<th>G977</th>
<th>G991</th>
<th>G996</th>
<th>G998</th>
<th>Pro 7</th>
<th>A31</th>
<th>A31'</th>
<th>A03s</th>
<th>A145</th>
<th>srsran</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size(MB)</td>
<td>41.2</td>
<td>41.8</td>
<td>41.5</td>
<td>41.6</td>
<td>44.0</td>
<td>44.3</td>
<td>44.3</td>
<td>66.6</td>
<td>66.3</td>
<td>66.3</td>
<td>17.8</td>
<td>22.5</td>
<td>22.5</td>
<td>16.8</td>
<td>17.0</td>
<td>92.9</td>
<td>43.0</td>
</tr>
<tr>
<td>Number of funcs</td>
<td>64K</td>
<td>61K</td>
<td>74K</td>
<td>74K</td>
<td>92K</td>
<td>75K</td>
<td>92K</td>
<td>103K</td>
<td>108K</td>
<td>103K</td>
<td>48K</td>
<td>94K</td>
<td>94K</td>
<td>65K</td>
<td>65K</td>
<td>96K</td>
<td>82K</td>
</tr>
<tr>
<td>Rank</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1.56</td>
<td></td>
</tr>
</tbody>
</table>

<The rank of the integrity protection function for each firmware>
Evaluation
How effectively can BaseComp discover bugs?

• Summary
  • 34 Mismatches
  • 29 True Positives
    • Classified to 15 types
  • 5 False Positives

<table>
<thead>
<tr>
<th></th>
<th>Samsung</th>
<th>MediaTek</th>
<th>srsRAN</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mismatches</td>
<td>9</td>
<td>10</td>
<td>15</td>
<td>34</td>
</tr>
<tr>
<td>False Positives</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>True Positives</td>
<td>8</td>
<td>7</td>
<td>14</td>
<td>29</td>
</tr>
</tbody>
</table>
Case Study
NAS AKA Bypass Vulnerability

- NAS Authentication and Key Agreement
Case Study

NAS AKA Bypass Vulnerability

• NAS Authentication and Key Agreement bypass
  • Attach Accept message to connect to malicious base station
  • Send arbitrary NAS messages in plaintext
  • Gather IMEI with Identity Request message
  • Modify time with EMM Information message
  • ...

Diagram:
- Victim
  - Attach Request
  - Attach Accept
  - Attach Complete
  - SMS message
- Attacker
  - Not Protected
  - AKA Bypass
  - Modified message (sec_hdr_t > 3)
Case Study
NAS AKA Bypass Vulnerability

- NAS Authentication and Key Agreement bypass

```c
bool IntegrityProtection(void* message) {
    int sec_state =_SECURE,
    uint8_t sec_hdr_type = GetSecHdrType(message);
    if (CheckHeader(message) != SECURE || sec_hdr_type == 0 || sec_hdr_type > 3)
        return false;
    if (IsProtected(message) || CheckSeq(message) || ValidateMac(message))
        return true;
    return false;
}
```

Message is accepted!

(regardless of any other element of the message)
Case Study
NAS AKA Bypass Vulnerability

- Delivering an arbitrary SMS message
  - Sender
    - 010-1000-1100
  - Time
    - January 3rd, 2030
  - SMS Data
    - Hello World!! from 2030
Conclusion

• Proposed a novel semi-automatic approach to analyze the integrity protection.
  • Probabilistic inference + Comparative analysis

• Found 29 bugs from Samsung, MediaTek and srsRAN images.
  • Including critical NAS AKA bypass vulnerabilities.
Thank You!