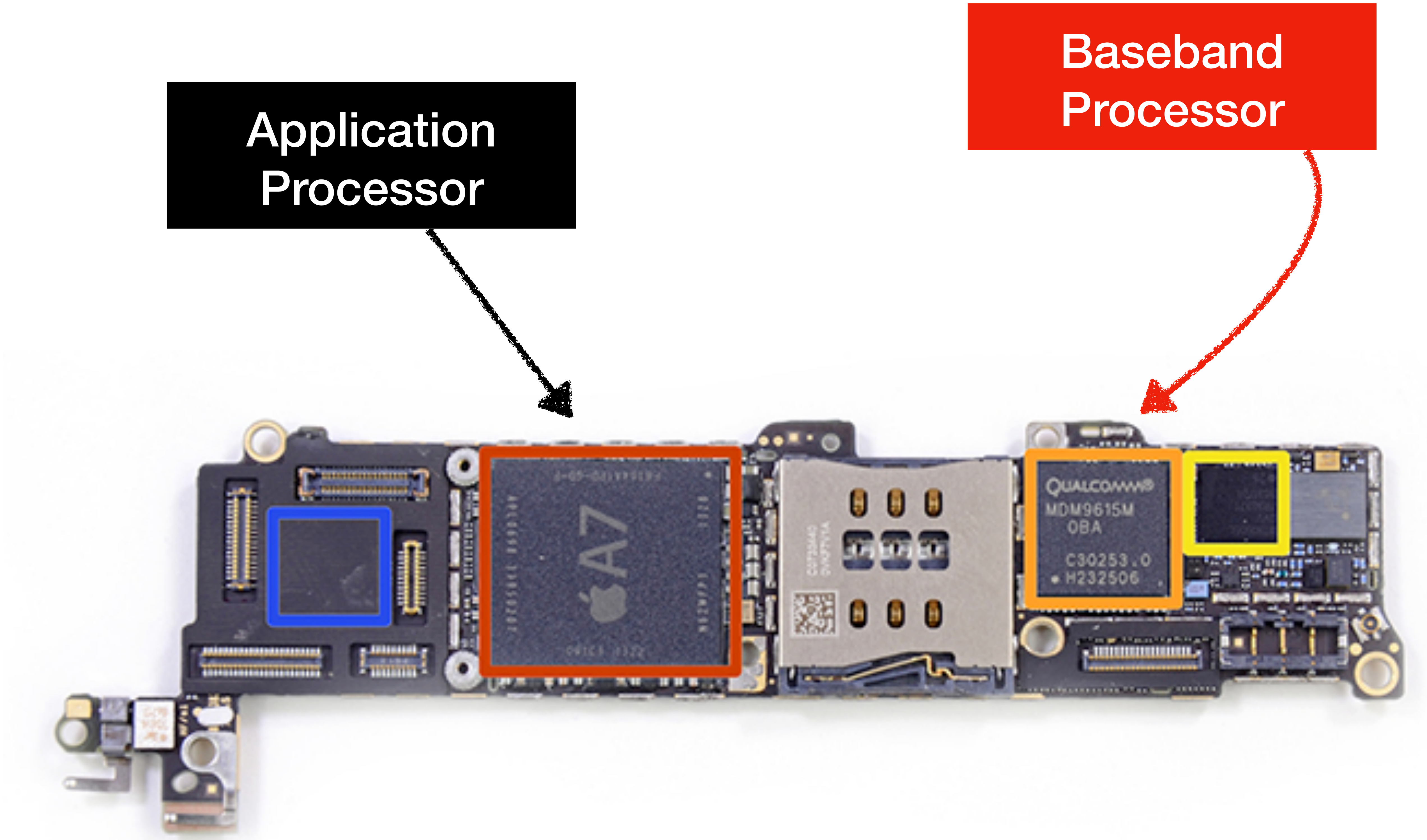
BaseComp: A Comparative Analysis for Integrity Protection in Cellular Baseband Software

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

†KAIST, ‡Samsung SDS

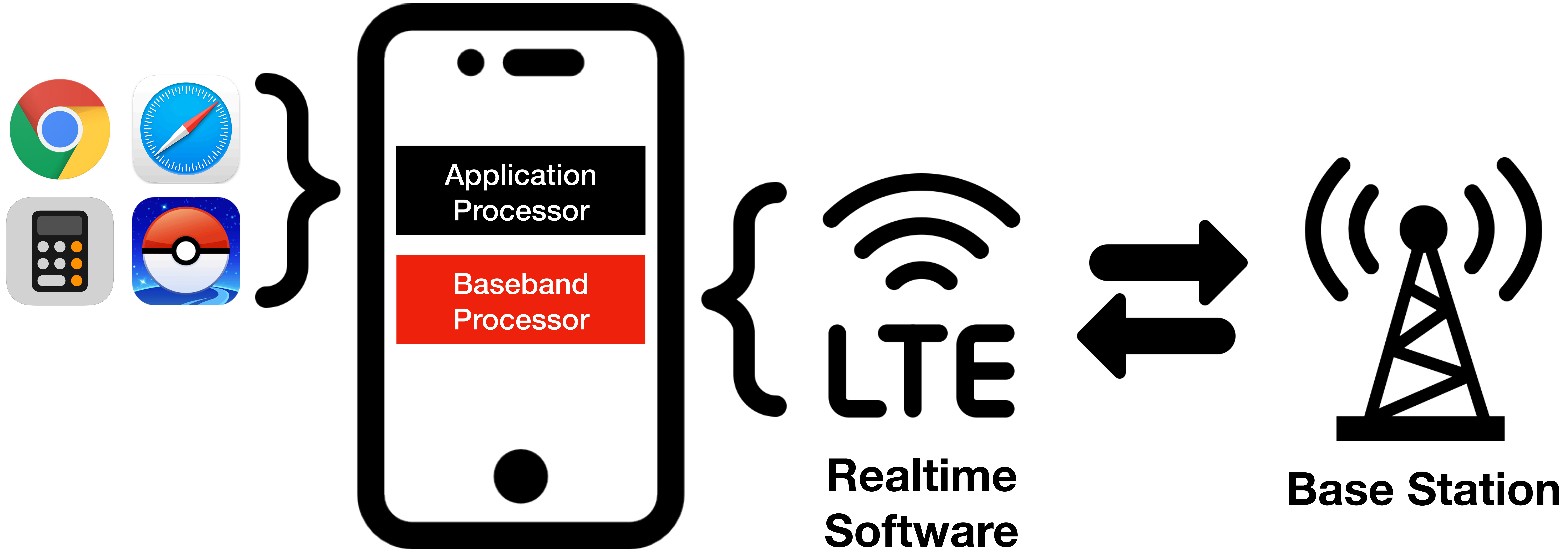
Baseband Software

Cellular Network Architecture



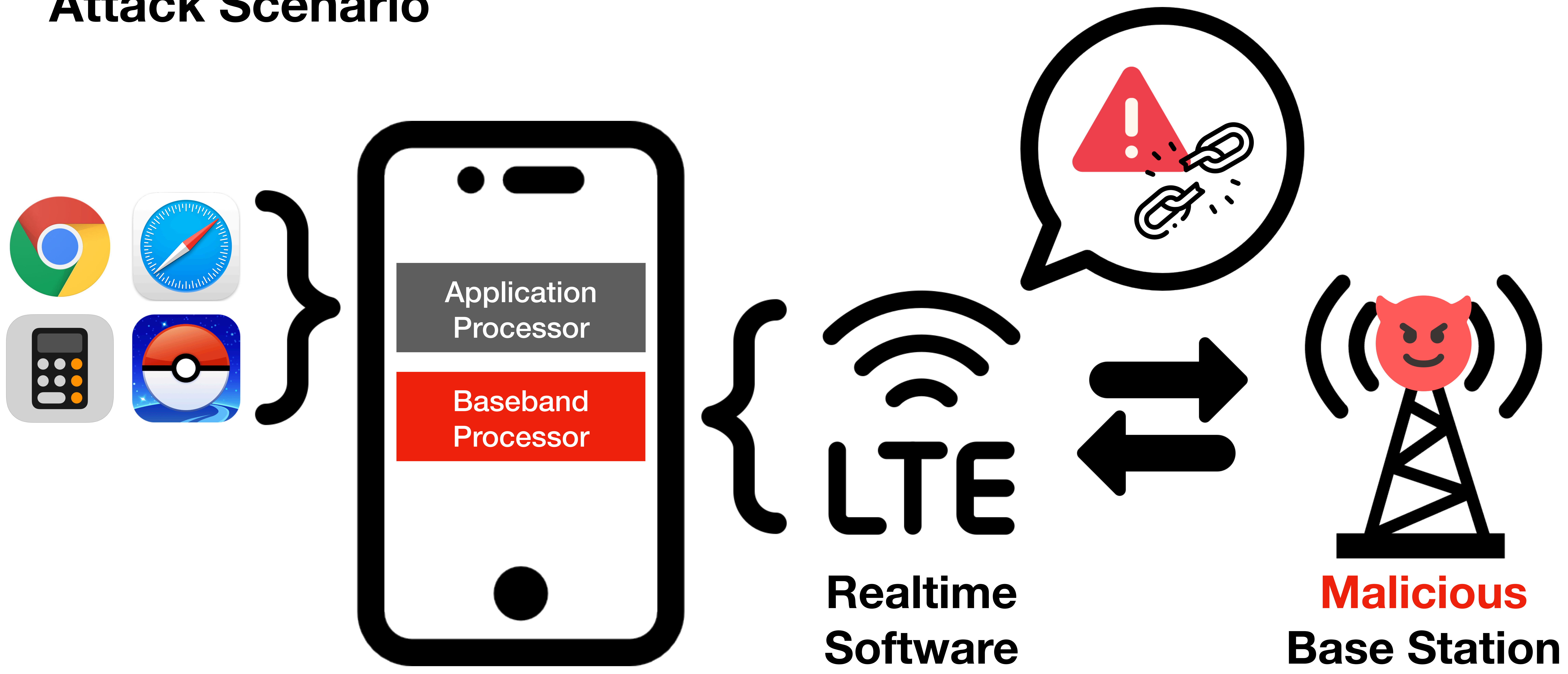
Baseband Software

Cellular Network Architecture



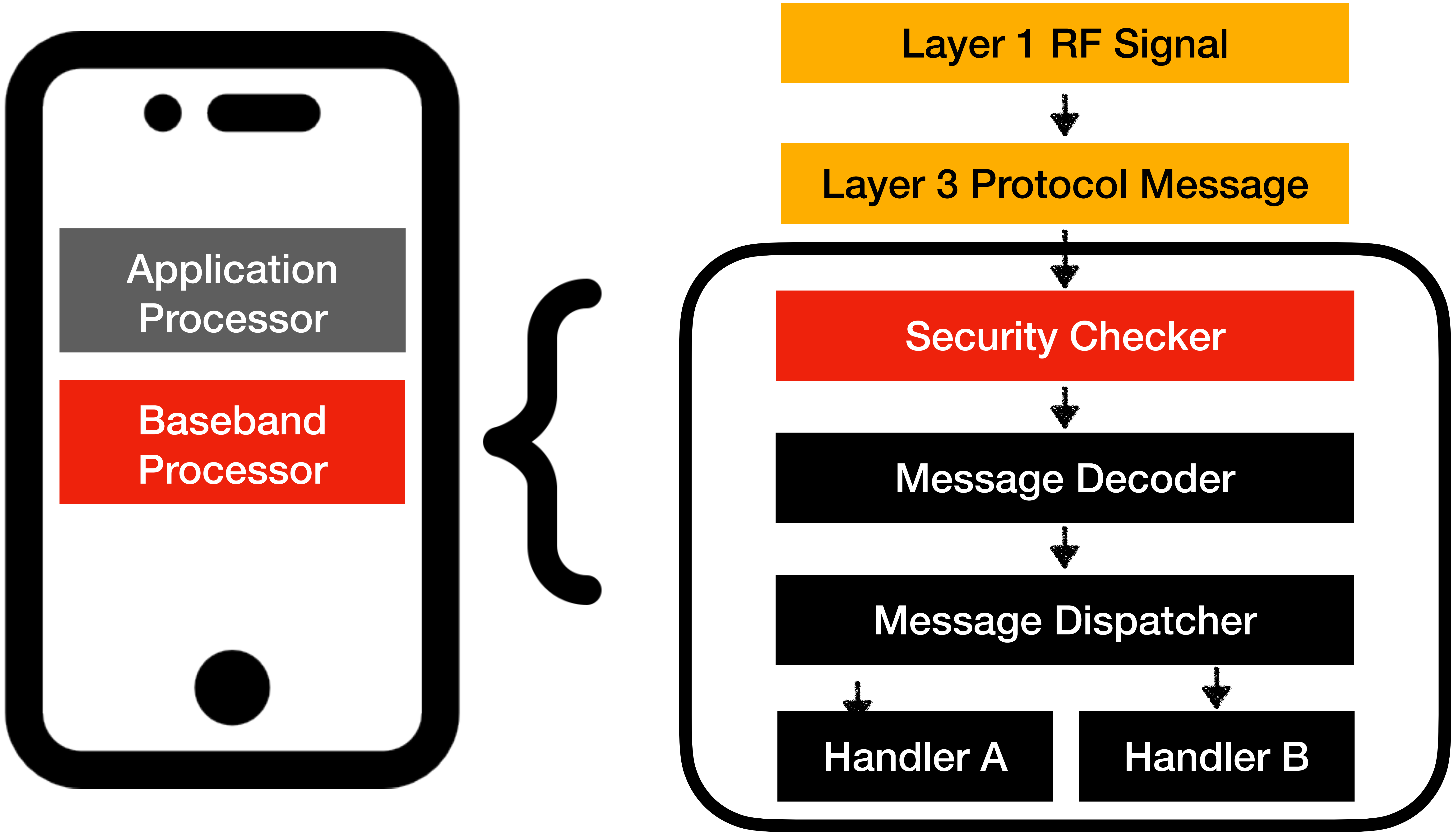
Baseband Software

Attack Scenario



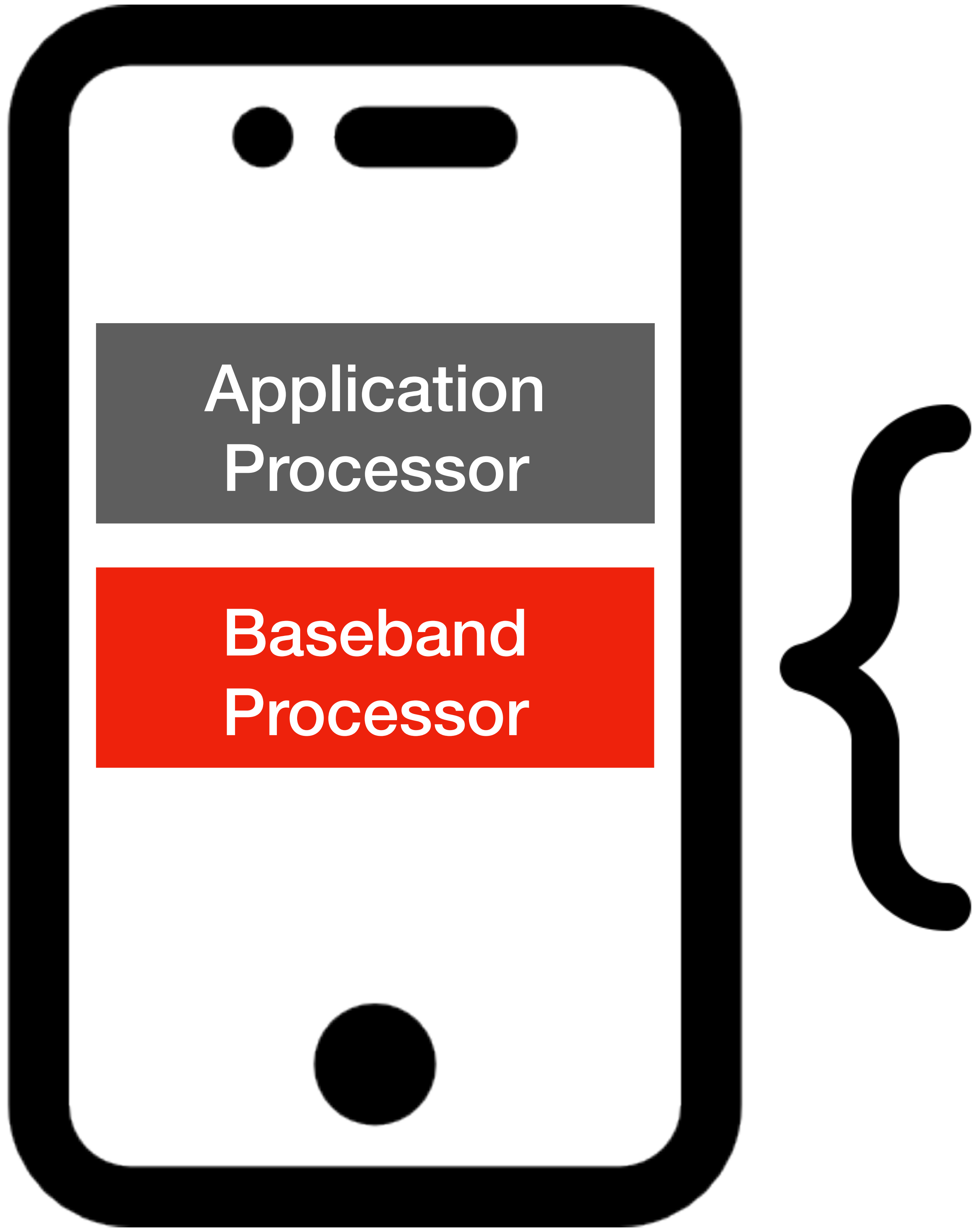
Baseband Software

Protocol Messages and Processing Logic

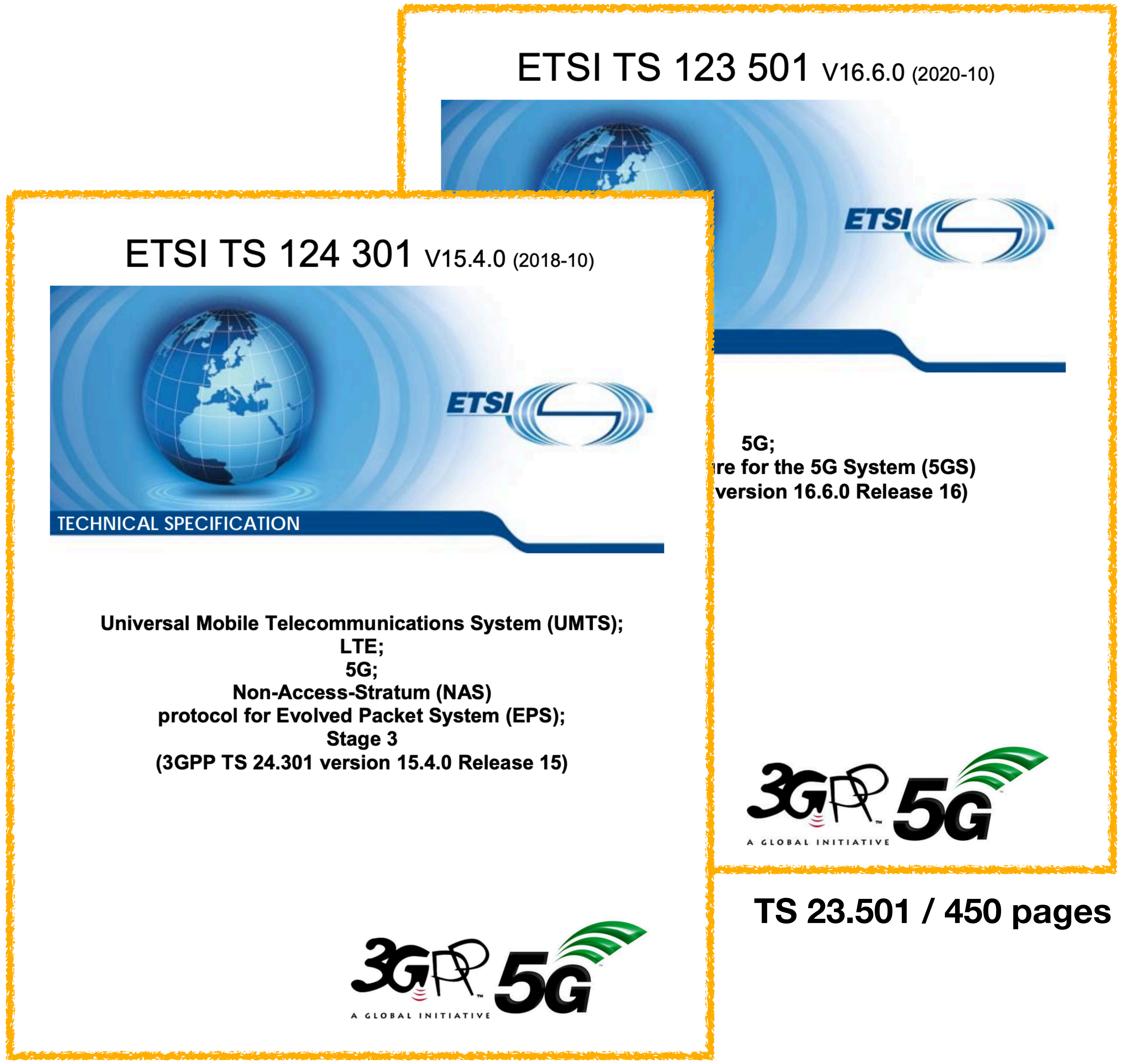


Baseband Software

Specification



3GPP (3rd Generation Partnership Project)

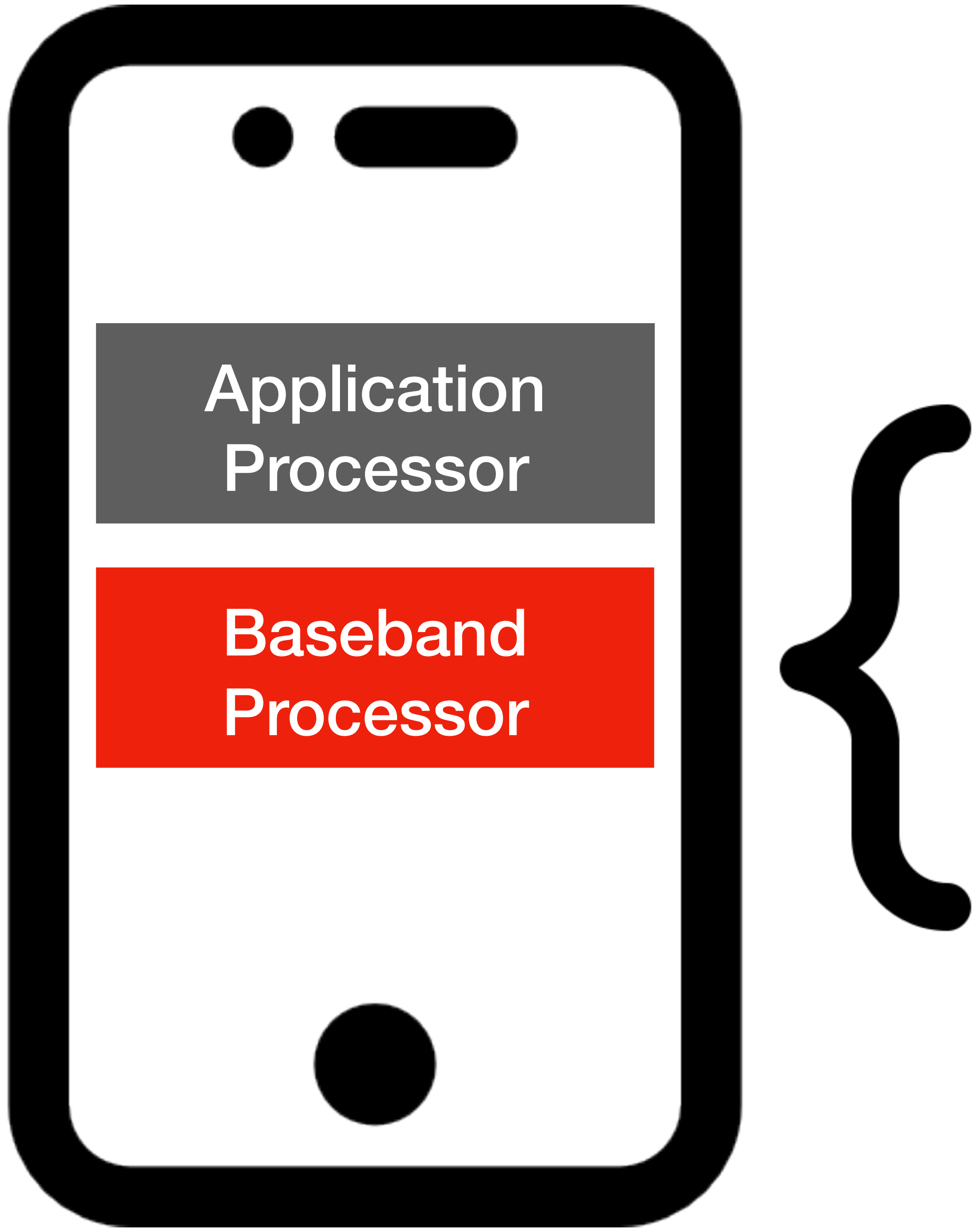


TS 23.501 / 450 pages

TS 24.301 / 500 pages

Baseband Software

Challenges



- **Large Binary Size**
  - The baseband software has to implement documents of n*100 pages
  - Average Binary Size: 43MB
  - Average # Functions: 83K
- **Obscurity**
  - Vendors don't release the details

Motivation

Existing Approaches (Related Work)

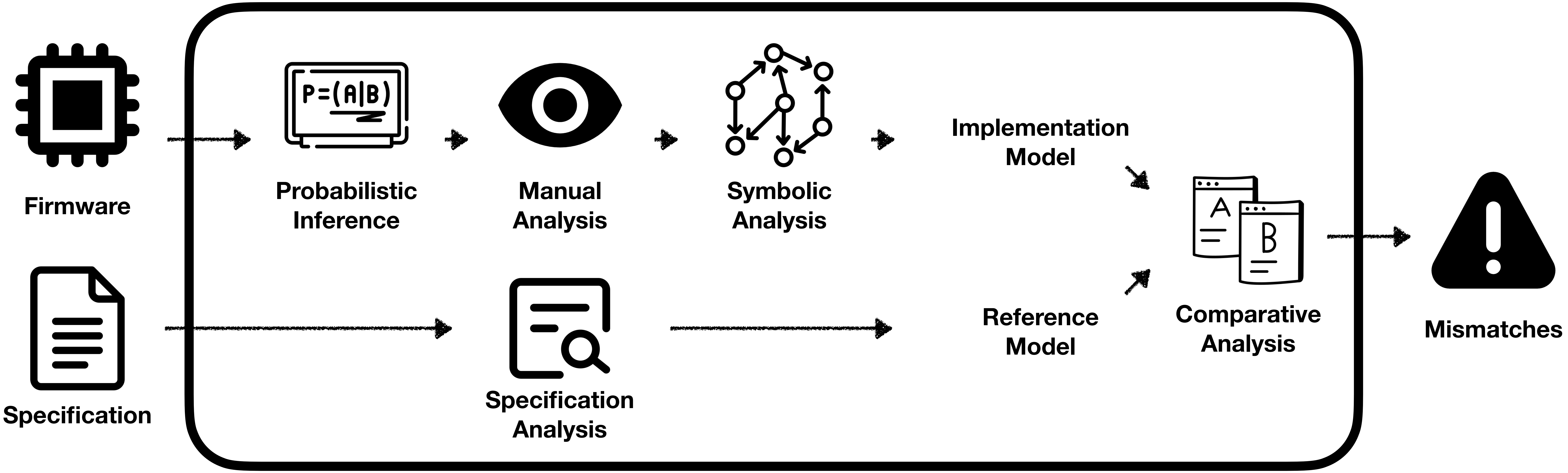
- **Dynamic Analysis**
  - DoLTEst (Security'22), Firmwire (NDSS'22)
  - Sends messages and observes responses from real or emulated devices
  - Has to restrict the search space leading to missing bugs
- **Static Analysis / BaseSpec (NDSS'21)**
  - Limited to message decoding and fails to analyze integrity protection
  - The vast size and obscurity causes highly resource-consuming manual analysis

Motivation

Our Approach

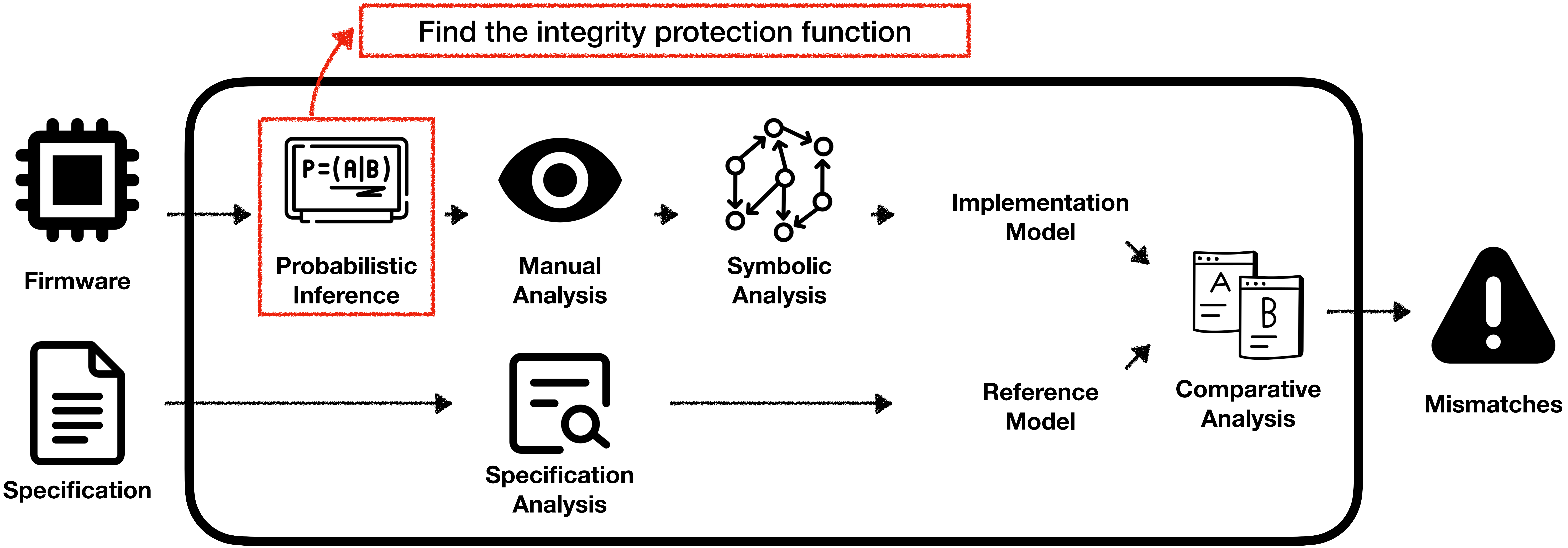
- **Static Analysis**
  - Without having to restrict the search space
- **Comparative Analysis**
  - Comparison with specification to uncover bugs in integrity protection
- **Probabilistic Inference**
  - Reduce the amount of manual effort needed

BaseComp
Overview



BaseComp

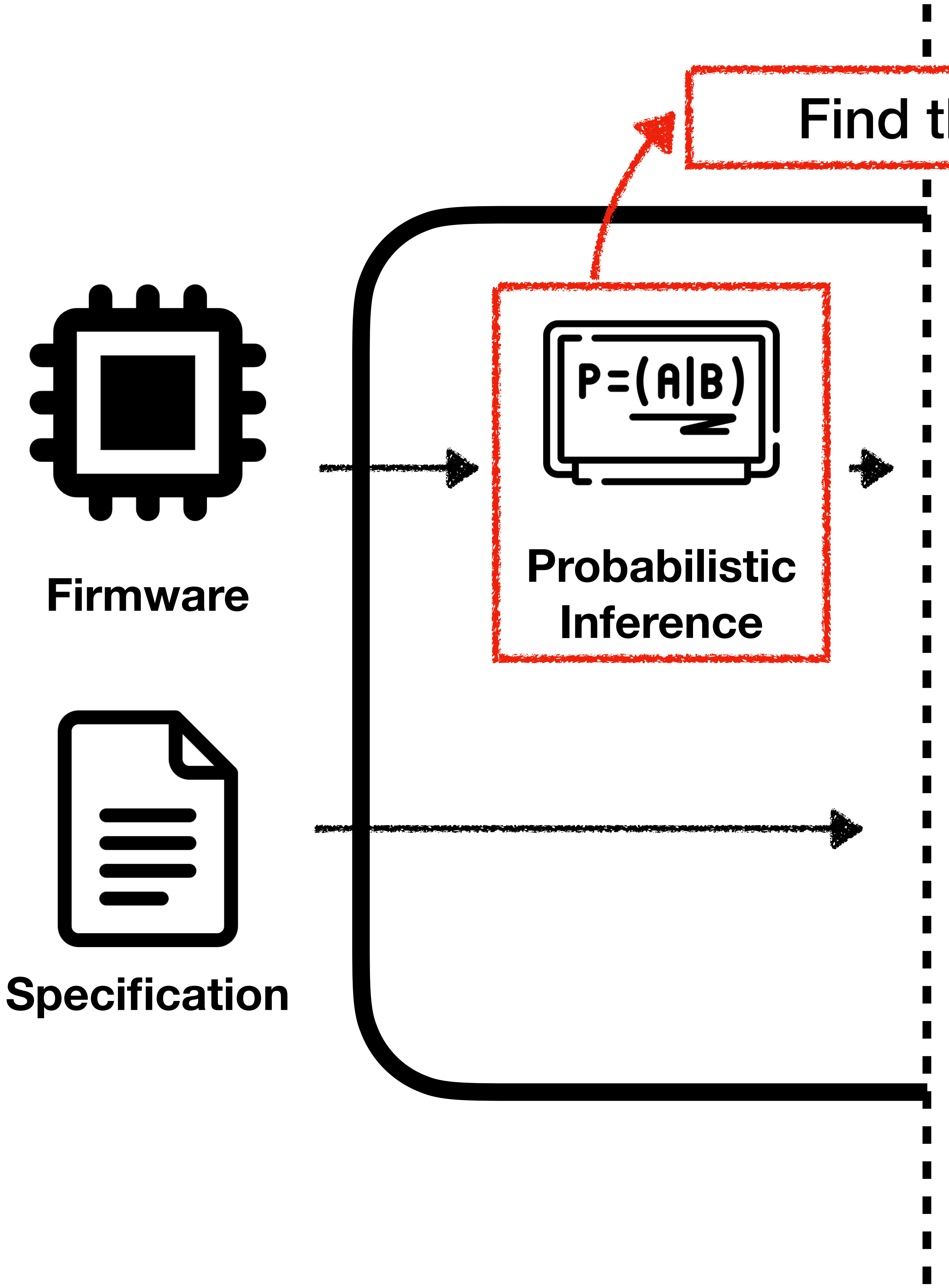
Probabilistic Inference



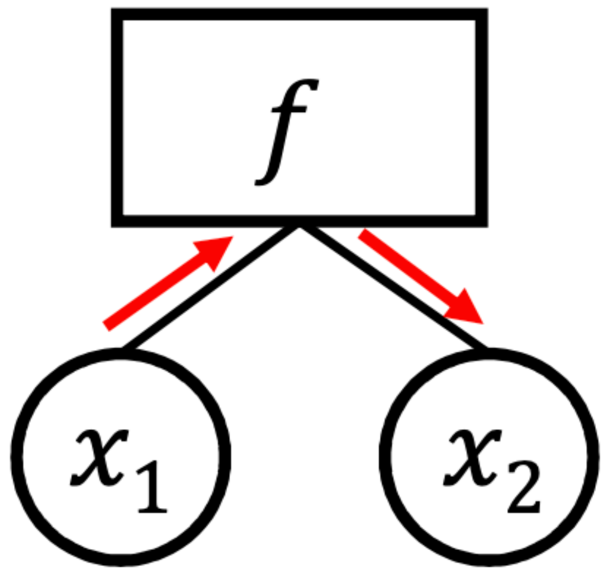
BaseComp

Probabilistic Inference

Find the integrity protection function



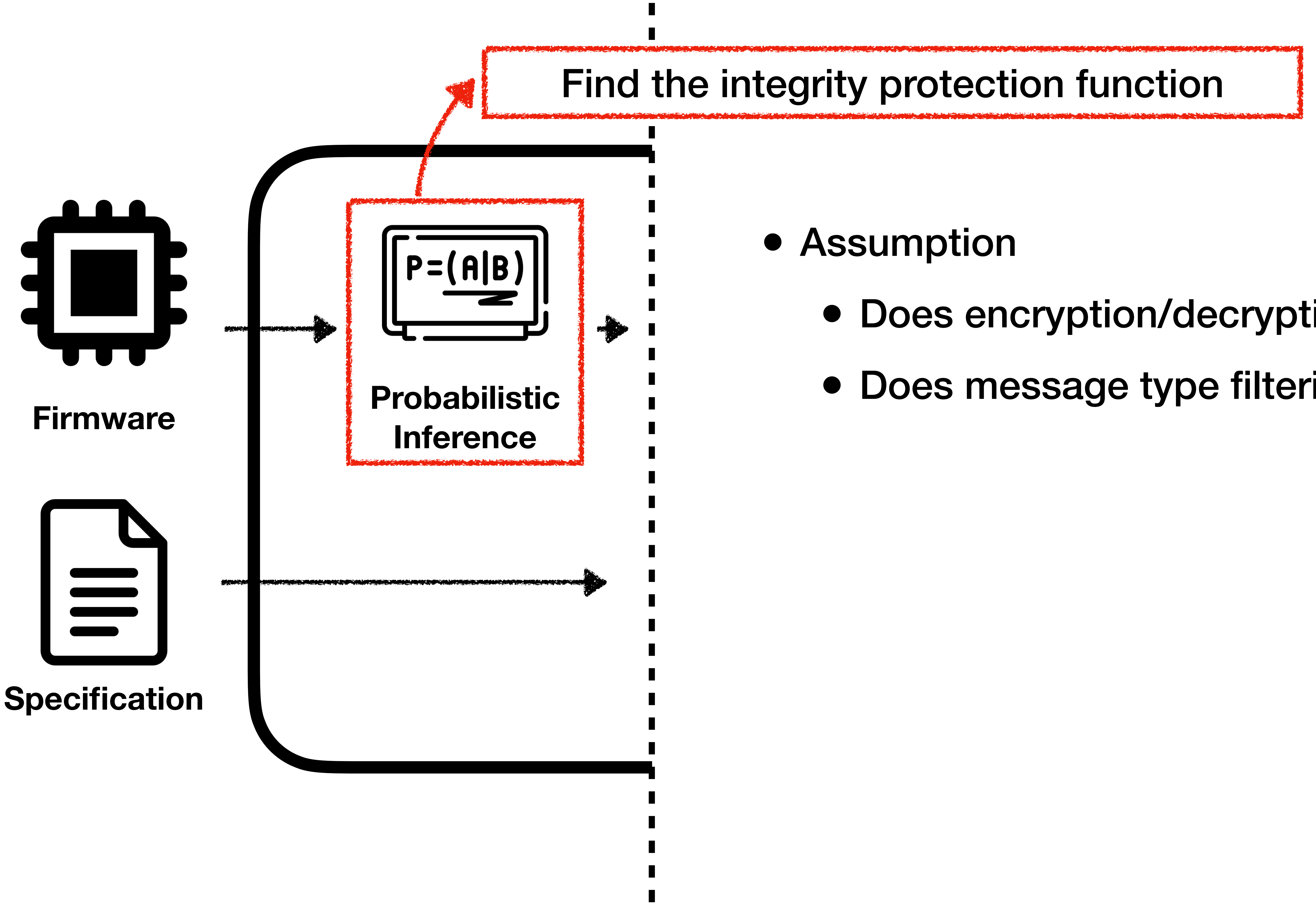
- Factor Graph
  - Variable Node: unknown quantities in problem
  - Function Node: function on subset of the variable nodes



		$x_1 \xrightarrow{Pos} x_2$	$x_1 \xrightarrow{Neg} x_2$	$x_1 \xrightarrow{Pref} x_2$
x_1	x_2	f_{Pos}	f_{Neg}	f_{Pref}
0	0	0.5	0.5	0.5
0	1	0.5	0.5	p_{Pref}
1	0	p_{Pos}	$1 - p_{Neg}$	$1 - p_{Pref}$
1	1	$1 - p_{Pos}$	p_{Neg}	0.5

BaseComp

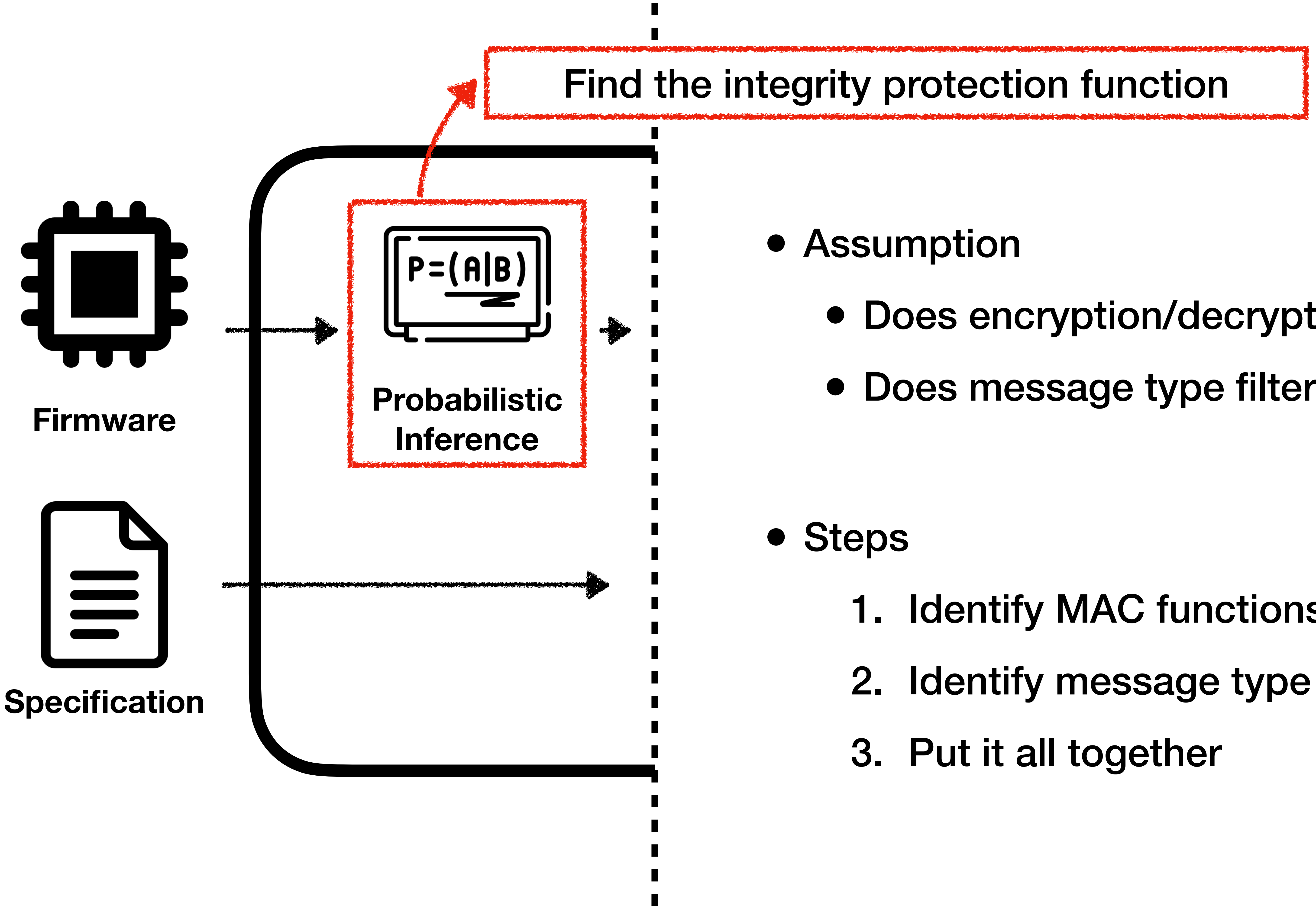
Probabilistic Inference



- Assumption
  - Does encryption/decryption using AES/ZUC/SNOW3G
  - Does message type filtering based on specification

BaseComp

Probabilistic Inference



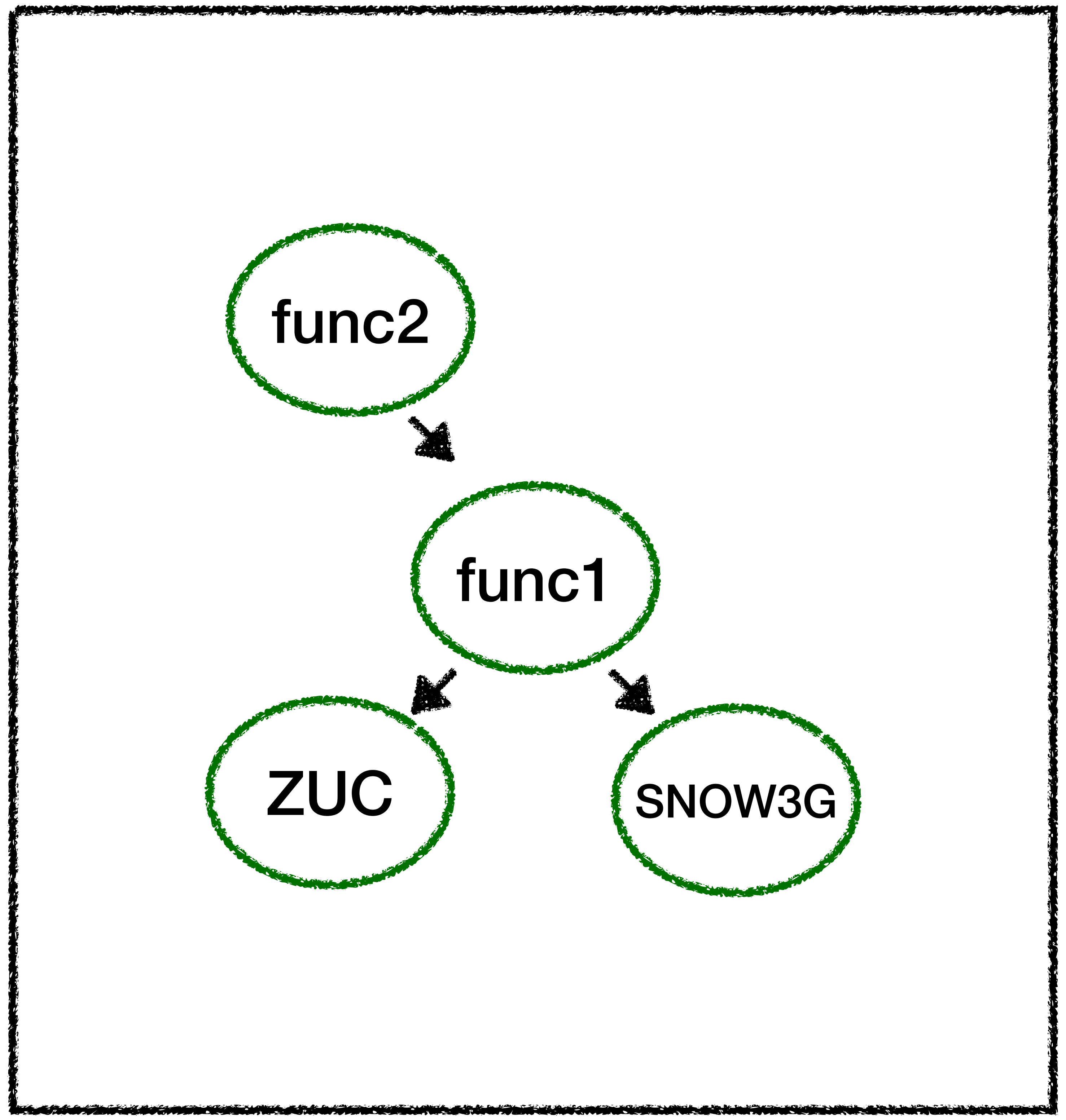
- Assumption
  - Does encryption/decryption using AES/ZUC/SNOW3G
  - Does message type filtering based on specification

- Steps
  1. Identify MAC functions
  2. Identify message type comparing functions
  3. Put it all together

BaseComp

Probabilistic Inference

1. Identify MAC functions



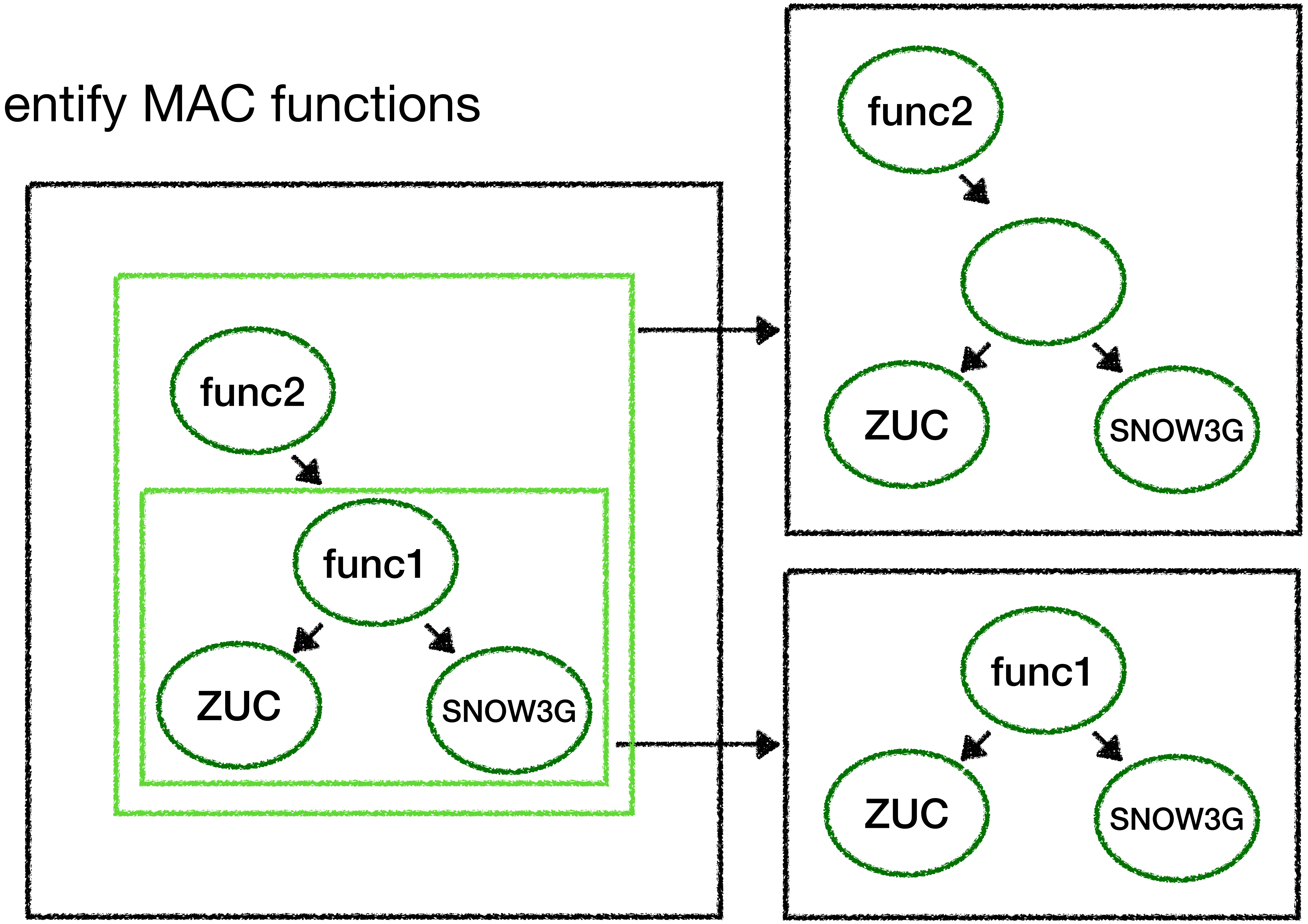
<Call Graph>

- Cryptographic functions identified by magic constants (S-Box)

BaseComp

Probabilistic Inference

1. Identify MAC functions



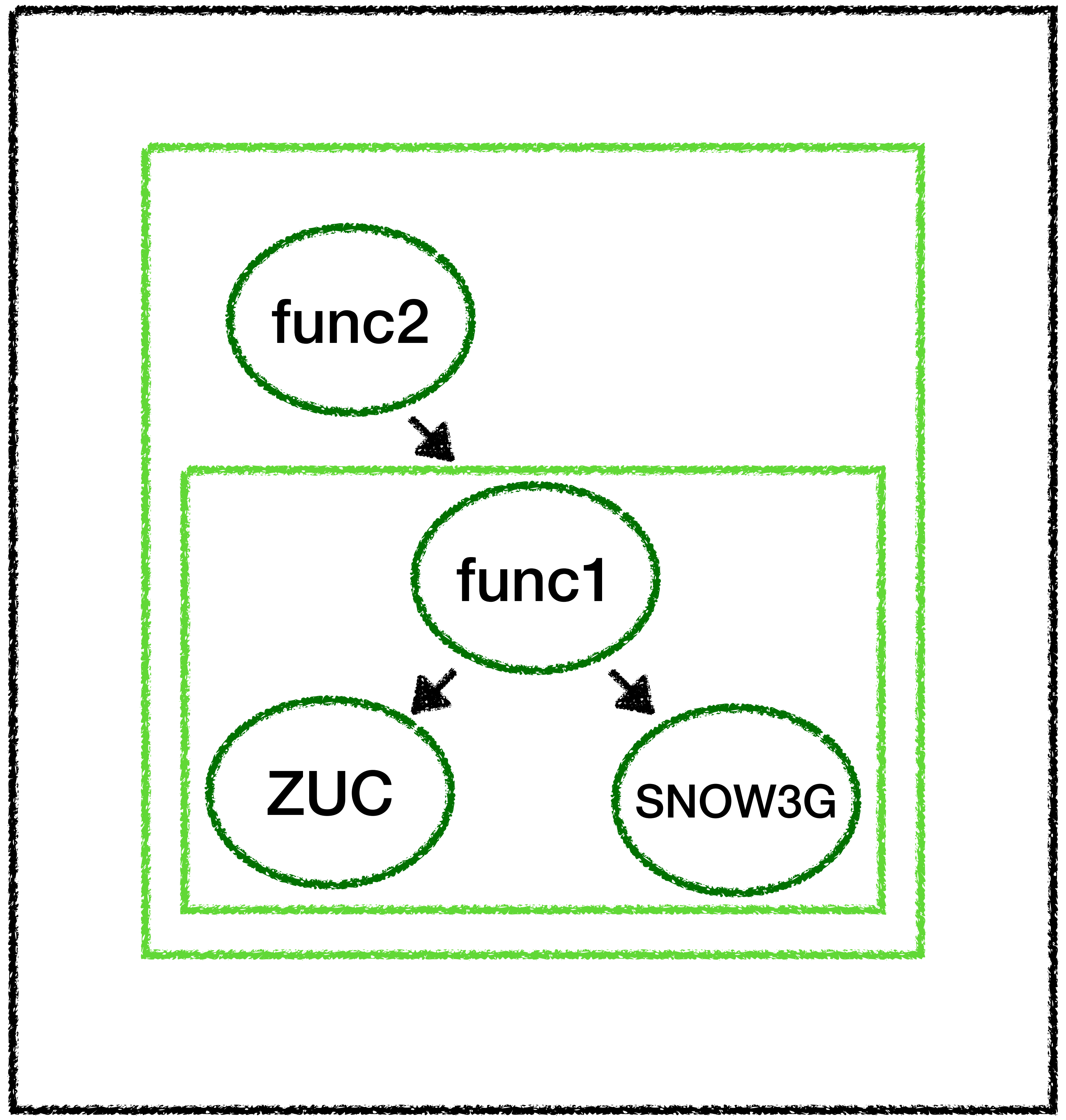
<Call Graph>

- Find common ancestors of cryptographic functions

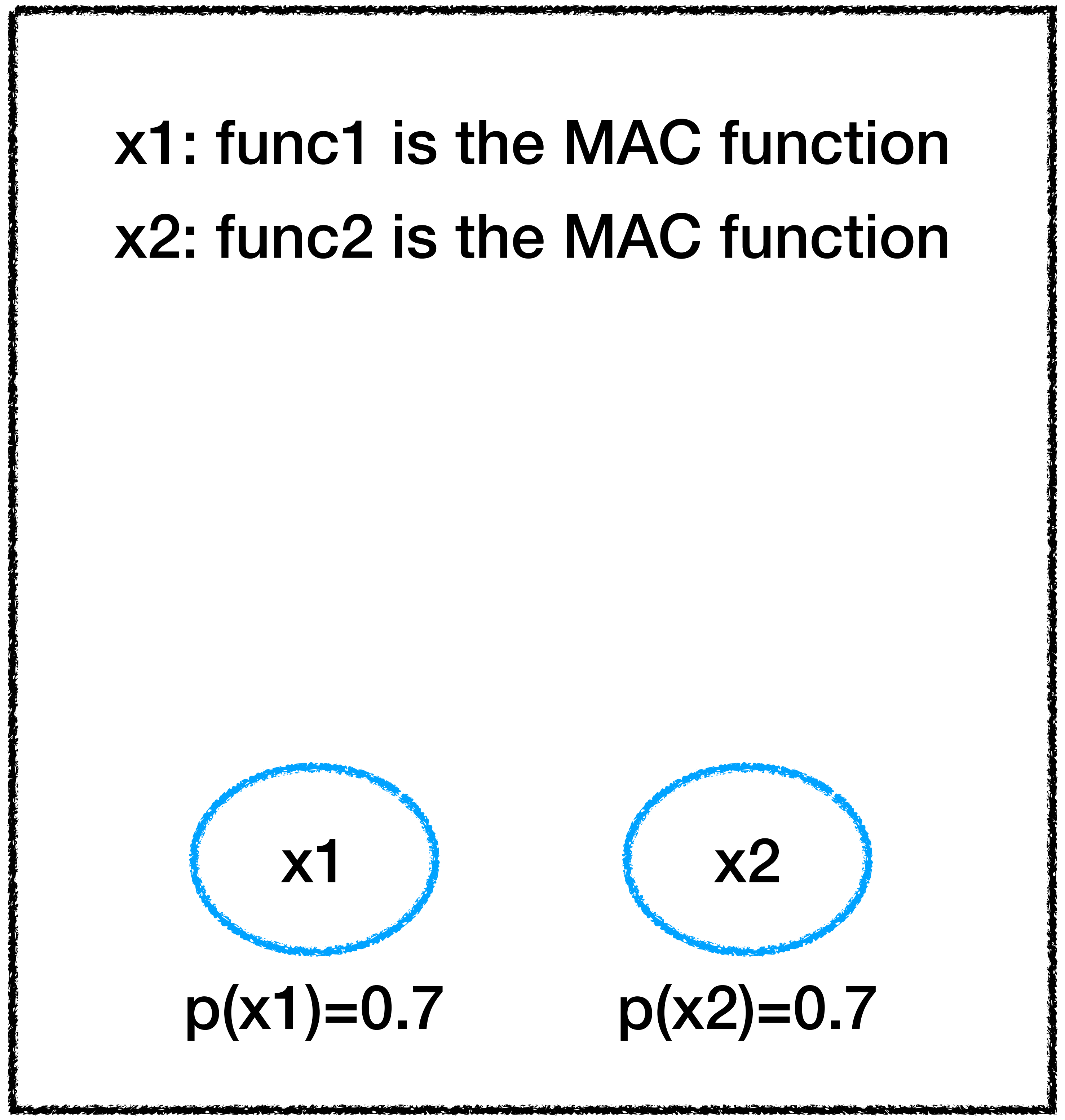
BaseComp

Probabilistic Inference

1. Identify MAC functions



<Call Graph>

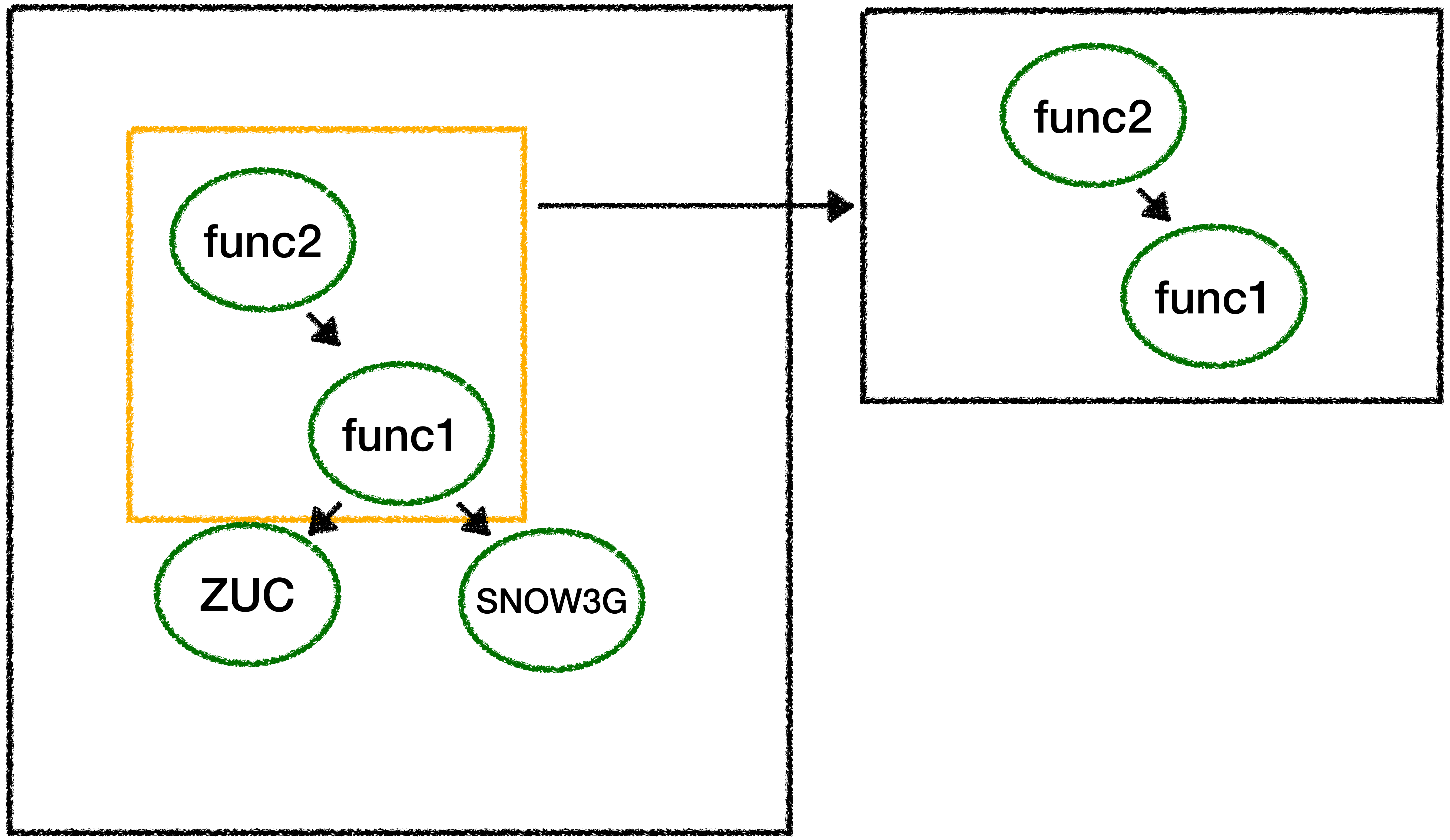


<Factor Graph>

BaseComp

Probabilistic Inference

1. Identify MAC functions



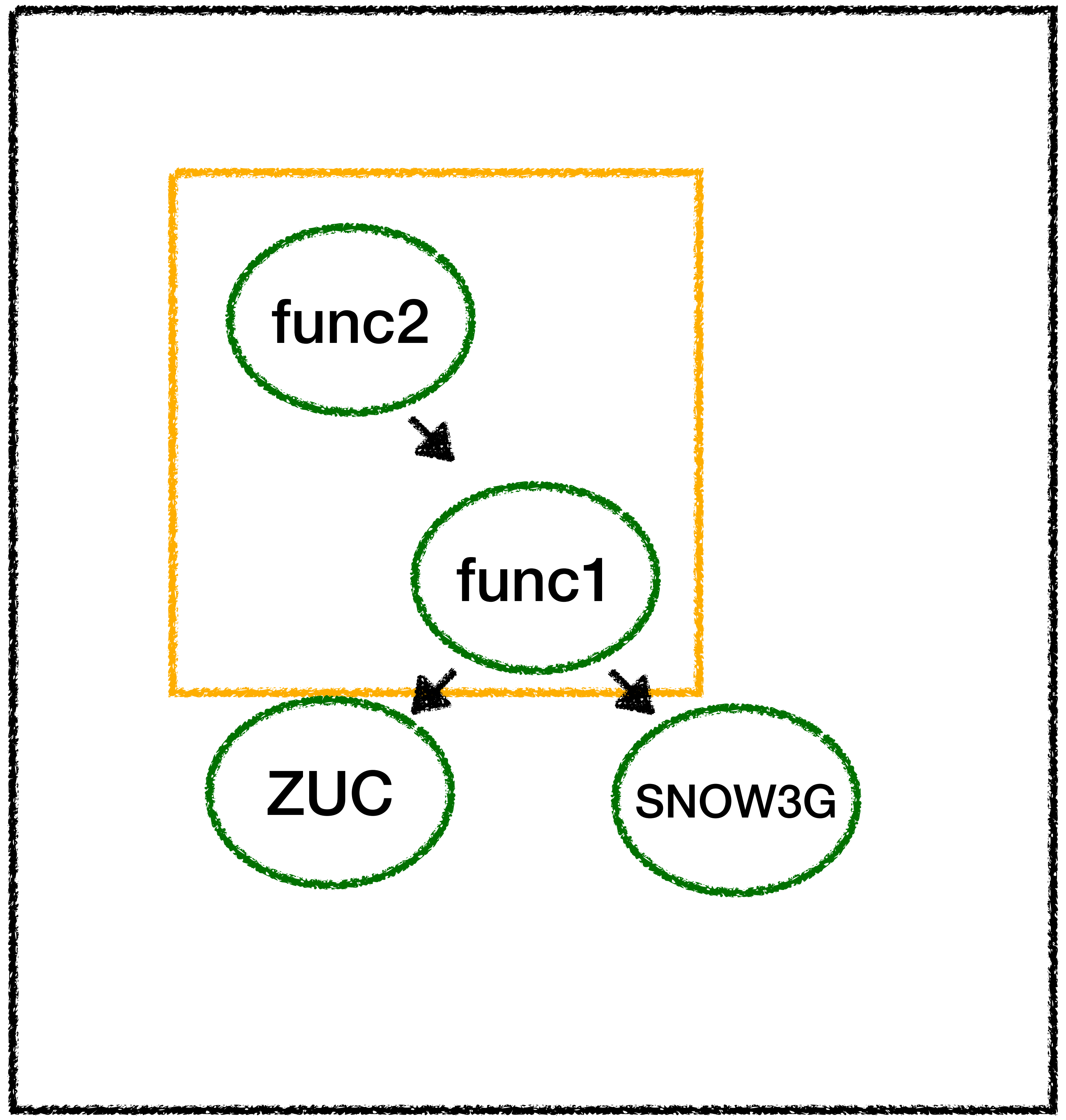
<Call Graph>

- Prioritize lower common ancestors

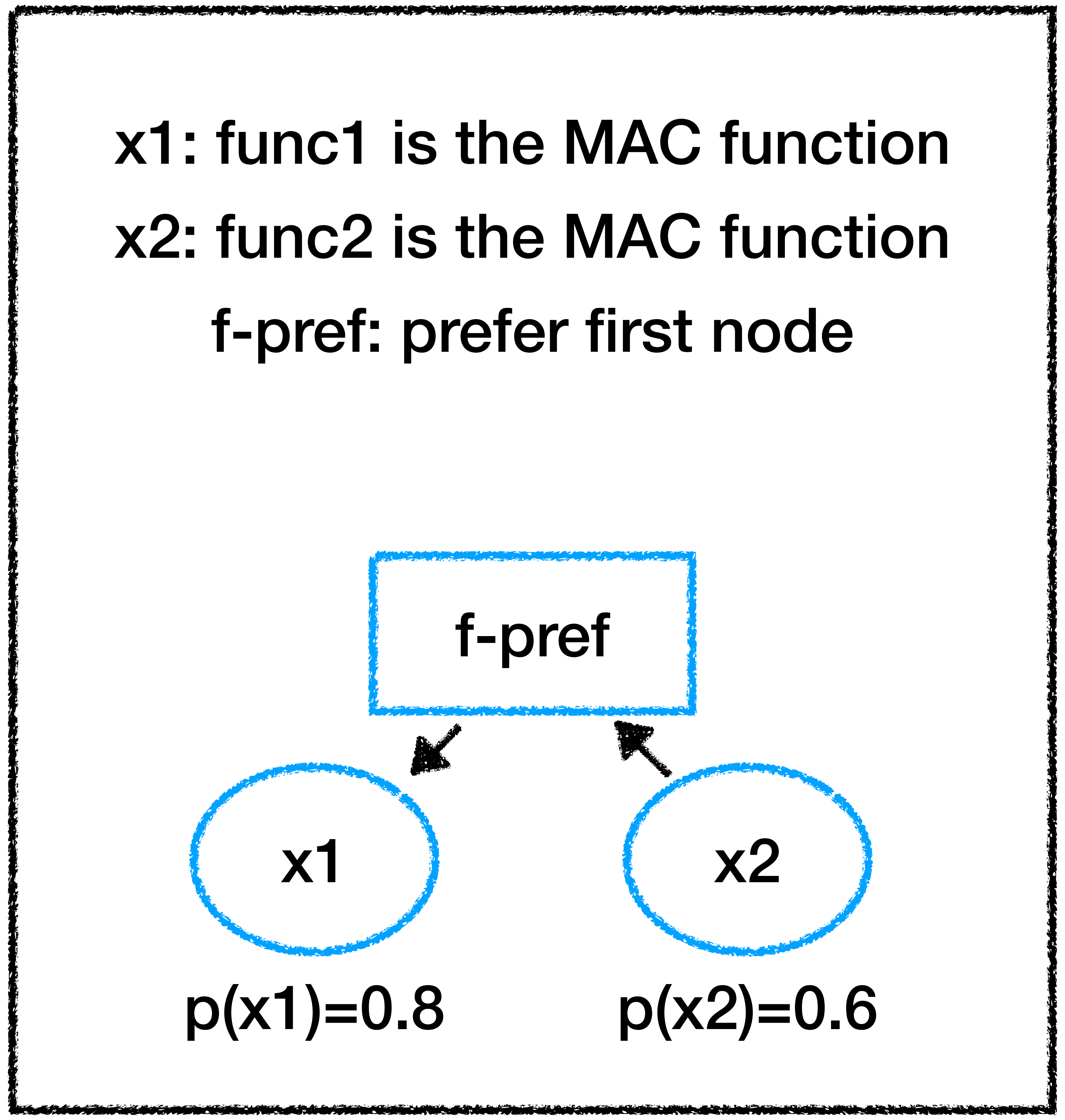
BaseComp

Probabilistic Inference

1. Identify MAC functions



<Call Graph>



<Factor Graph>

BaseComp

Probabilistic Inference

2. Identify 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).

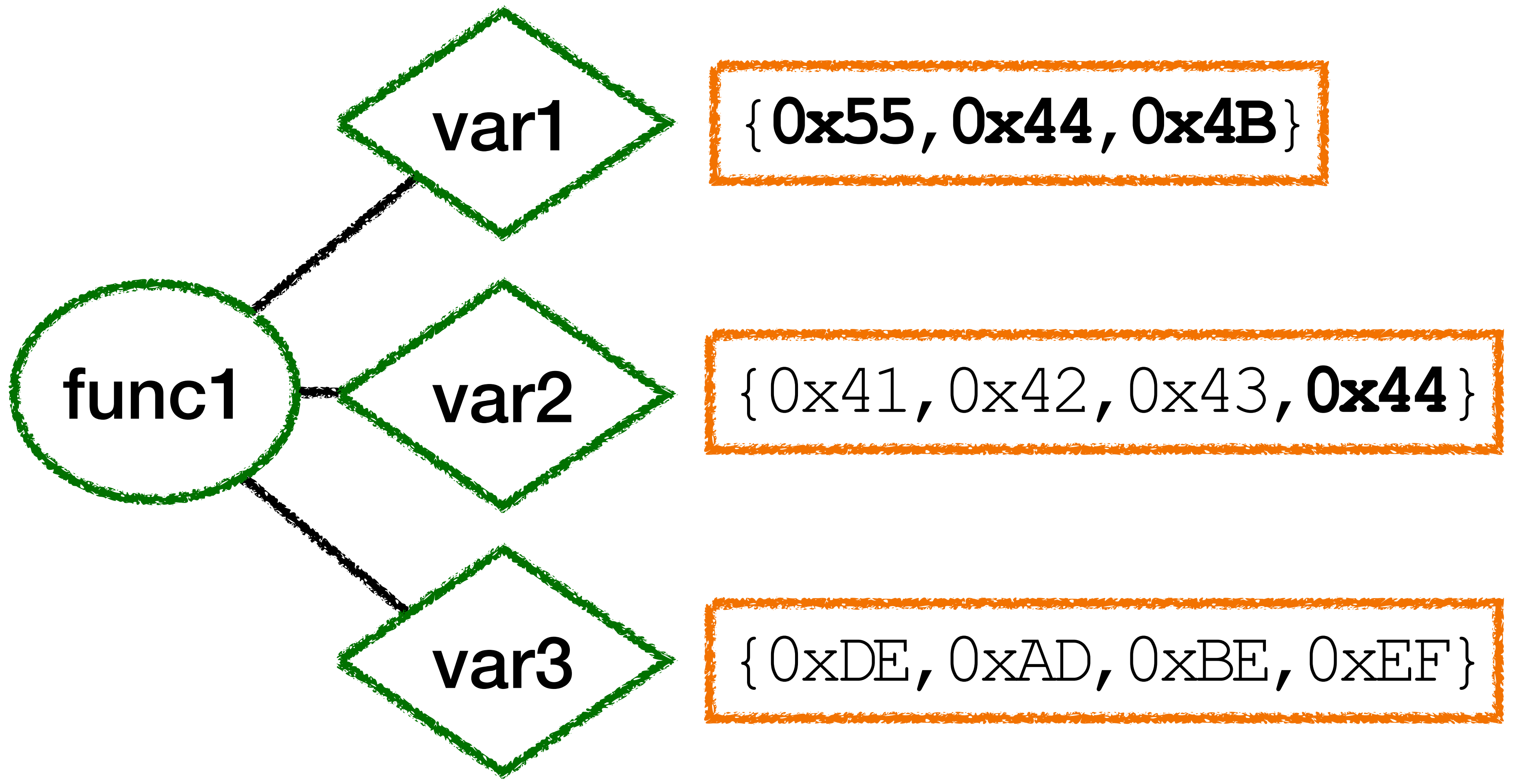
NOTE: These messages are accepted by the UE without integrity protection, as in certain situations they are sent by the network before security can be activated.

```
{0x55, 0x44, 0x4B, 0x4E, 0x52, 0x54, 0x46}
```

BaseComp

Probabilistic Inference

2. Identify message type comparing functions

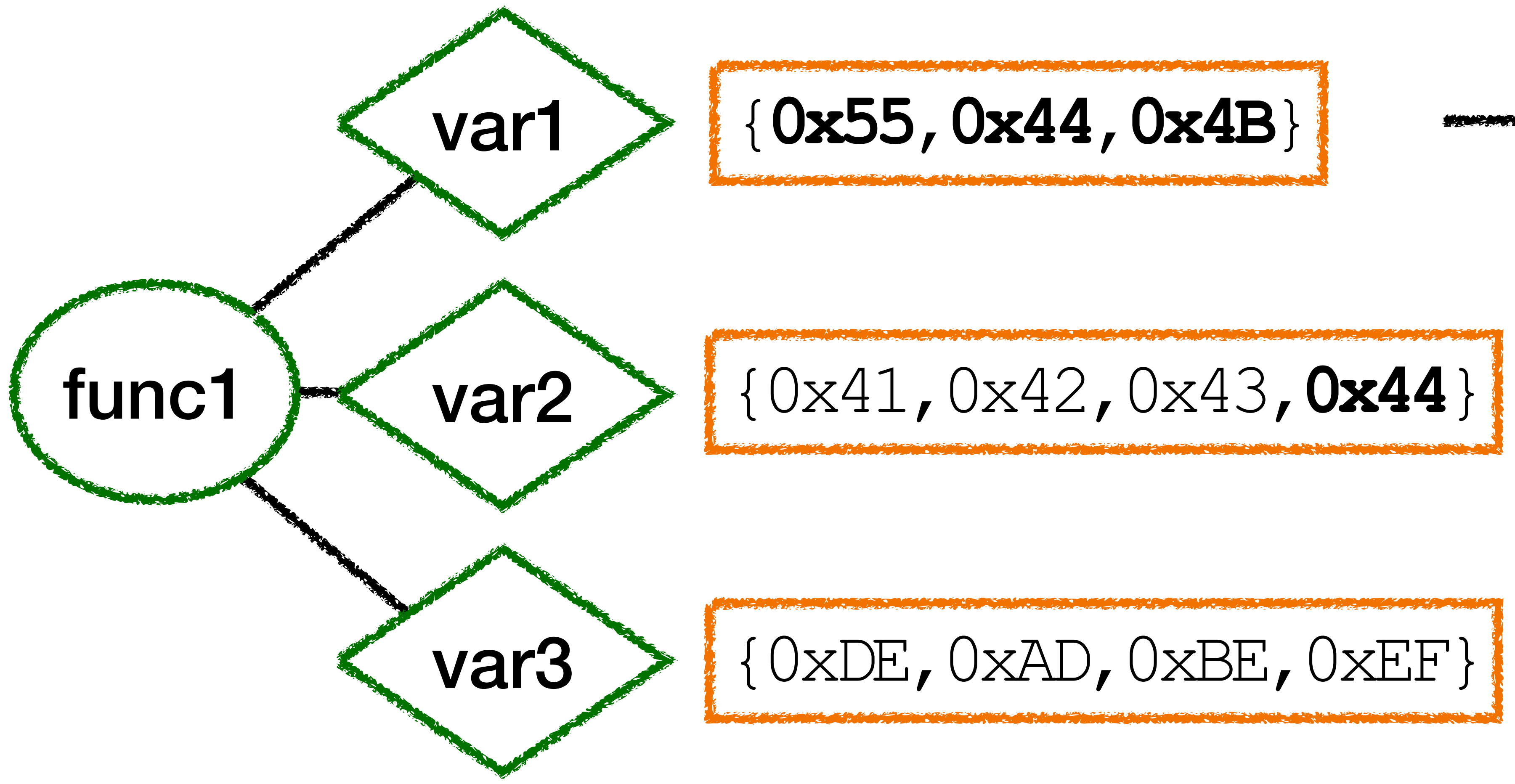


<Constants used for comparison with variables>

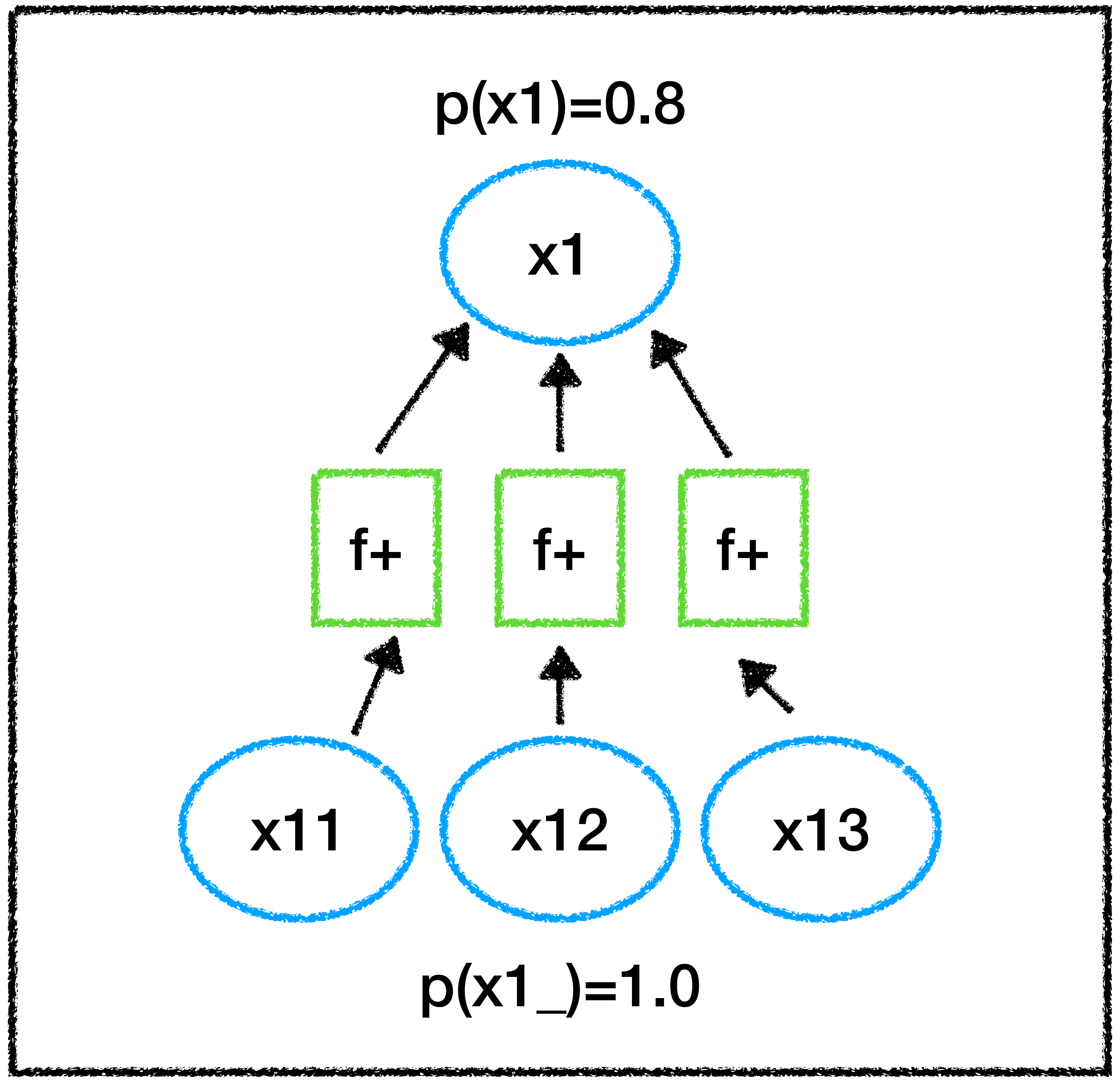
BaseComp

Probabilistic Inference

2. Identify message type comparing functions



<Constants used for comparison with variables>

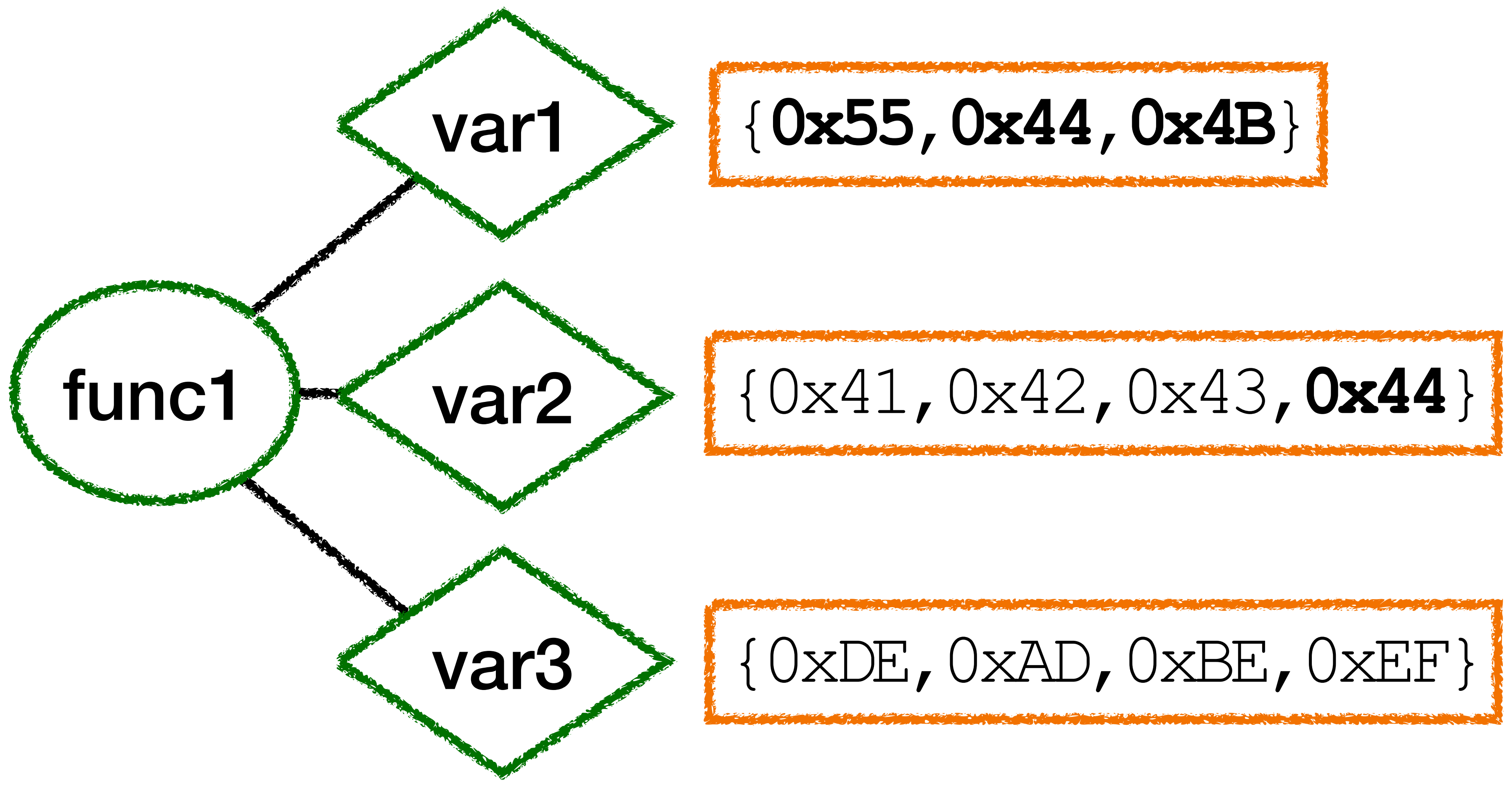


<Factor Graph for var1>

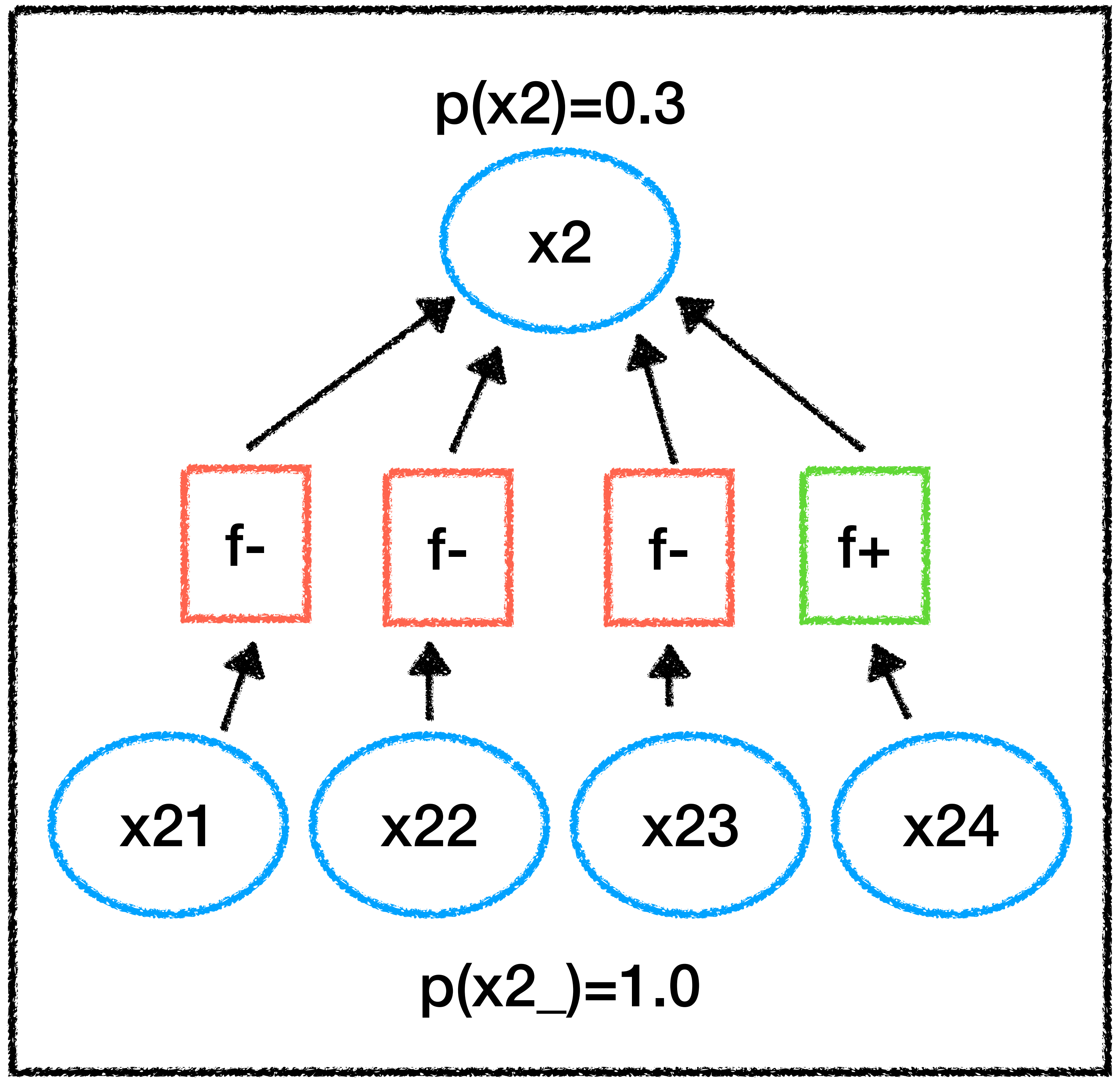
BaseComp

Probabilistic Inference

2. Identify message type comparing functions



<Constants used for comparison with variables>

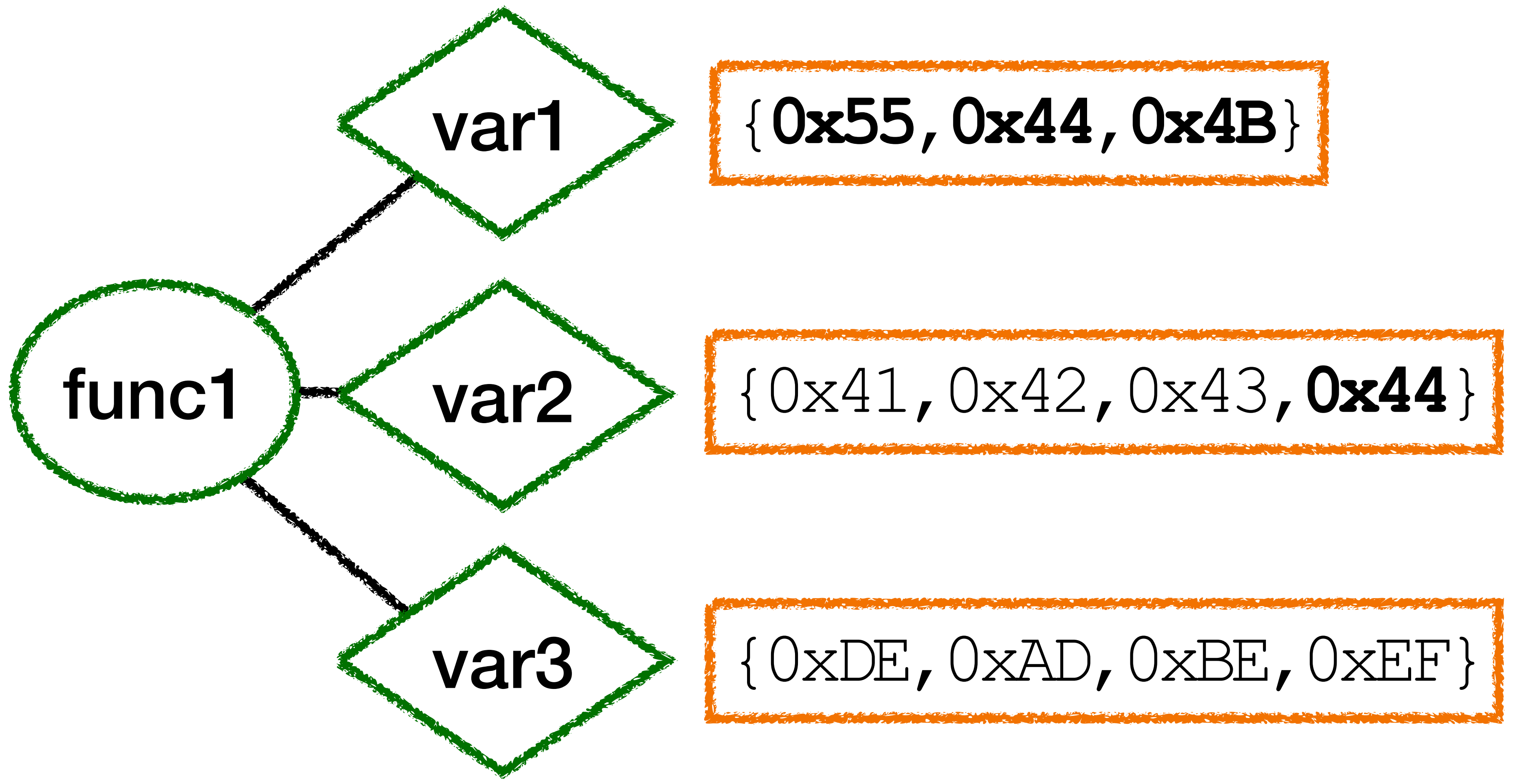


<Factor Graph for var2>

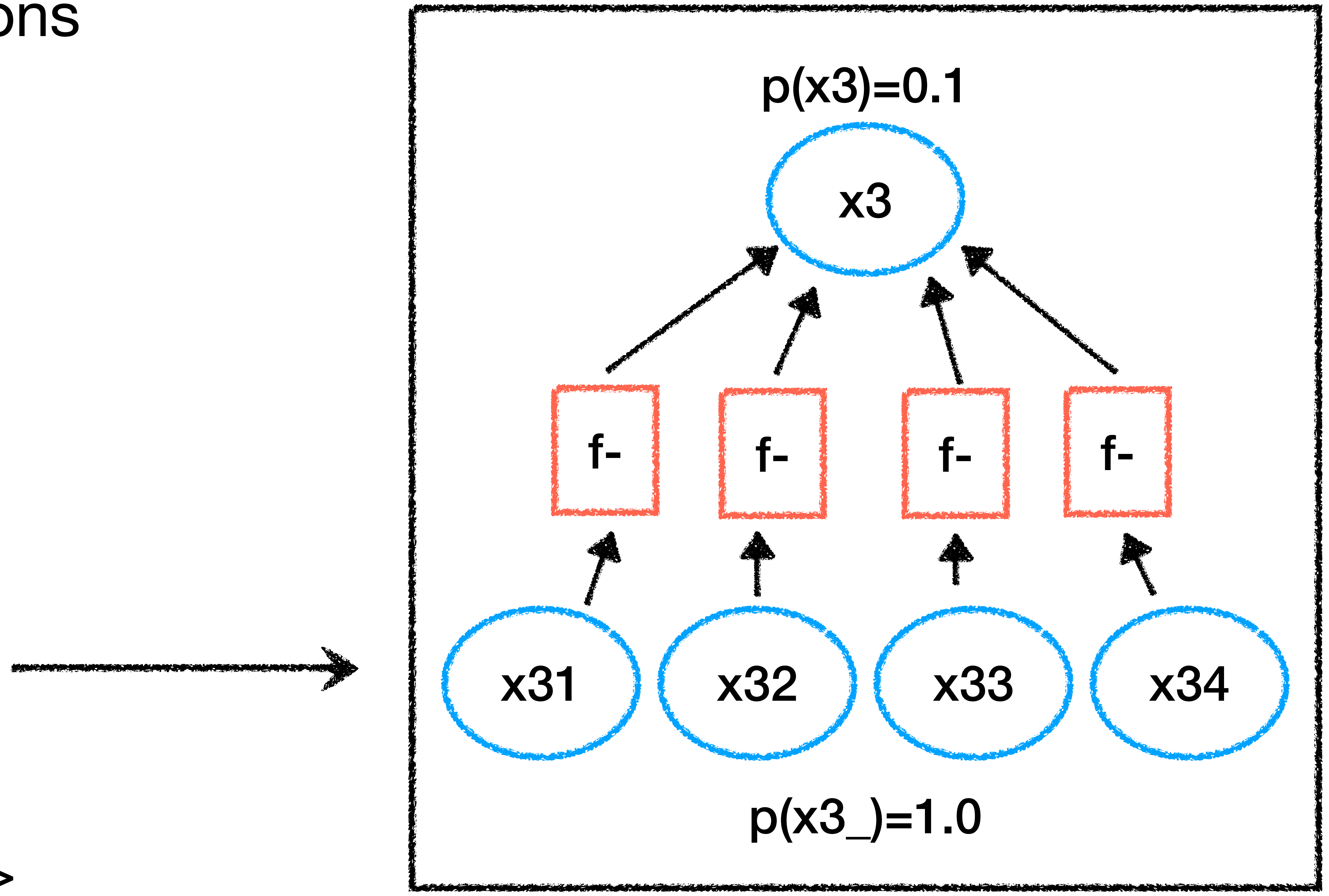
BaseComp

Probabilistic Inference

2. Identify message type comparing functions



<Constants used for comparison with variables>

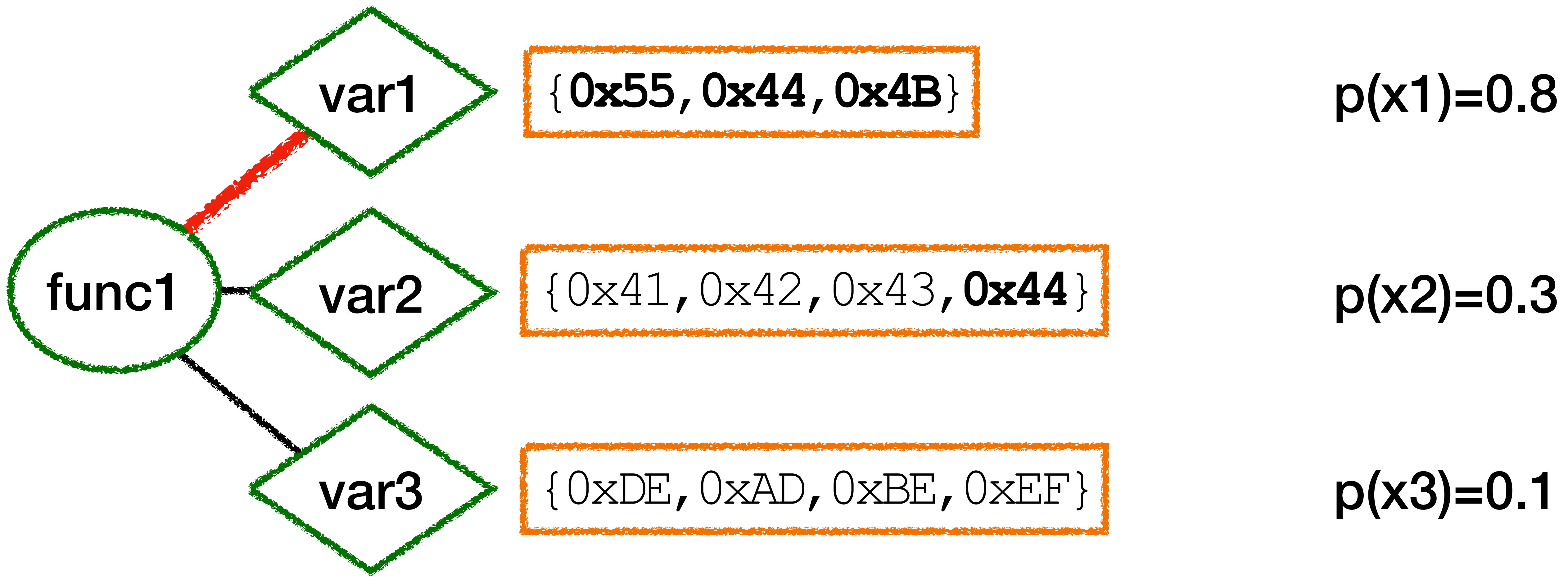


<Factor Graph for var3>

BaseComp

Probabilistic Inference

2. Identify message type comparing functions

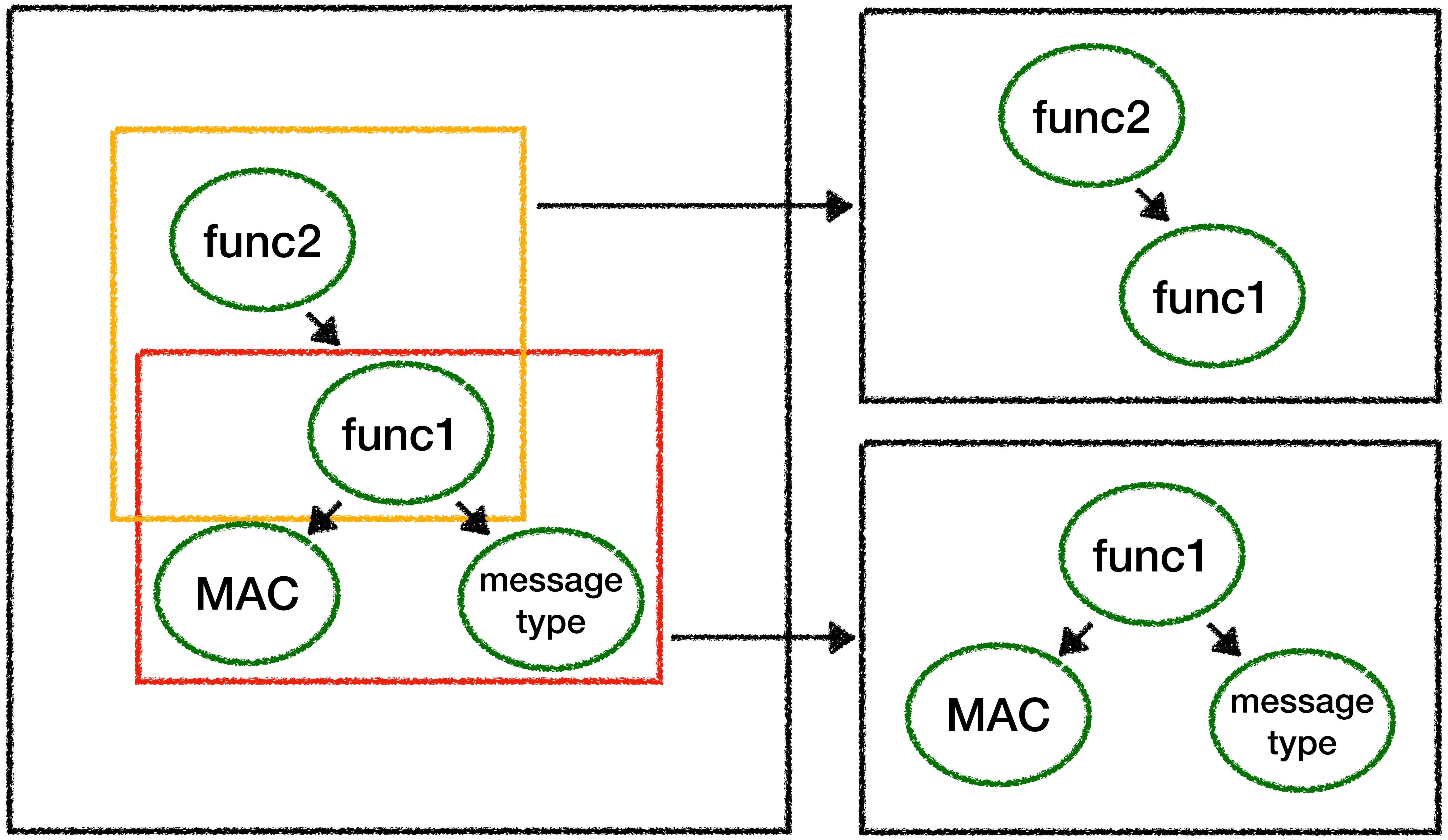


<Constants used for comparison with variables>

BaseComp

Probabilistic Inference

3. Put it all together

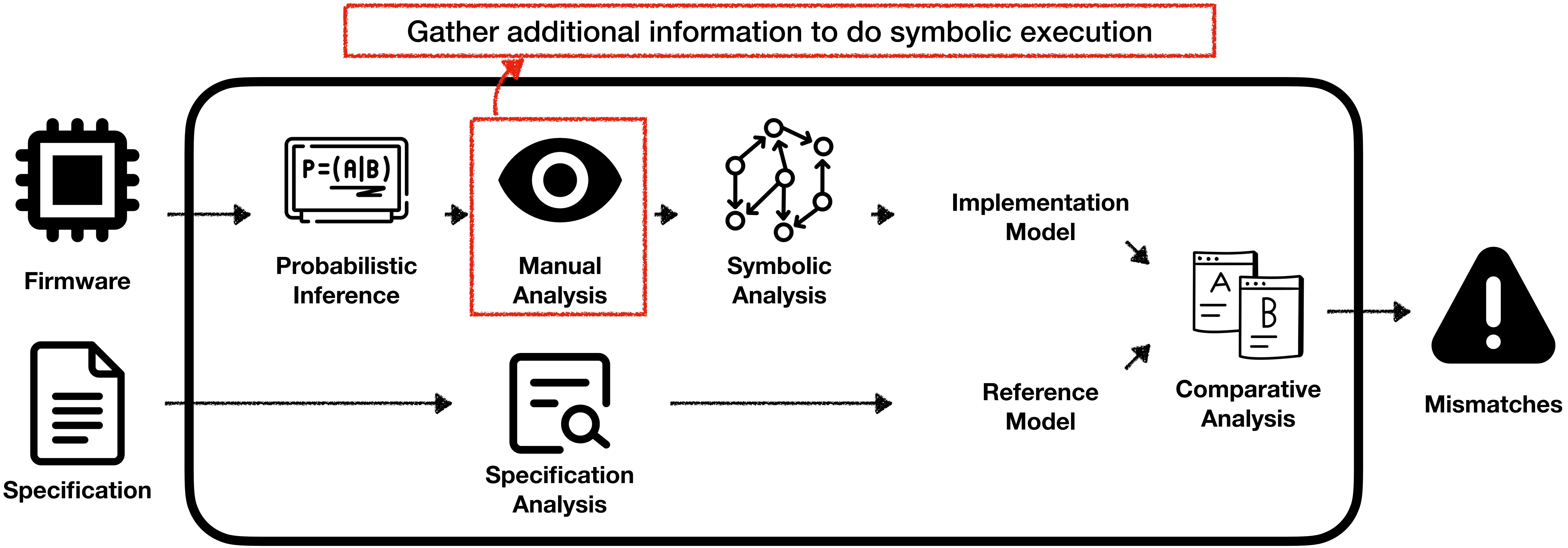


<Call Graph>

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

BaseComp

Manual Analysis



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

```
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

```
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**

```
analysis:           ./analysis_samsung.py

# Functions for analysis
integrity_func:         0x4150AECD
mac_validation_func:    0x4150A3D6
security_state:         0x429B27C4

# Functions to skip to avoid path explosion
skip_funcs:
    - 0x40CECC87
    - 0x4057F5FB
```

- **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

```
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**

```
analysis:          ./analysis_samsung.py

# Functions for analysis
integrity_func:          0x4150AECD
mac_validation_func:     0x4150A3D6
security_state:          0x429B27C4

# Functions to skip to avoid path explosion
skip_funcs:
    - 0x40CECC87
    - 0x4057F5FB
```

- **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
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
```

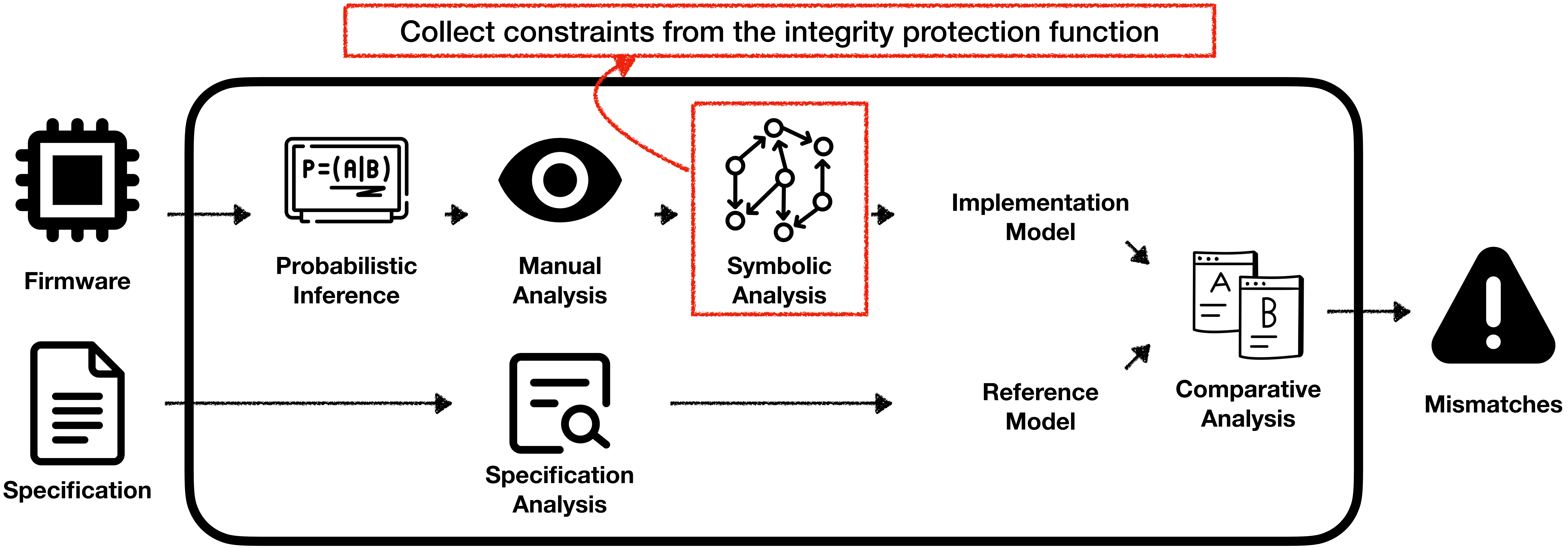
- **Vendor-specific analysis module**
  - How to symbolize variables
  - How to decide if a message is accepted
- **Required per-vendor**

```yaml
analysis:          ./analysis_samsung.py

# Functions for analysis
integrity_func:         0x4150AECD
mac_validation_func:    0x4150A3D6
security_state:         0x429B27C4

# Functions to skip to avoid path explosion
skip_funcs:
    - 0x40CECC87
    - 0x4057F5FB
```

- **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

Symbolic Analysis



BaseComp

Symbolic Analysis

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

```
// 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 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

```
// 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 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

sec_state == SECURE

+

0 < sec_hdr_type <= 3

```
// 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 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

sec_state == SECURE
+
0 < sec_hdr_type <= 3

sec_state != SECURE
+
sec_hdr_type == 0
+
msg_type in {0x55, ... , 0x46}

```
// 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 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

sec_state == SECURE
+
0 < sec_hdr_type <= 3

sec_state != SECURE
+
sec_hdr_type == 0
+
msg_type in {0x55, ... , 0x46}

+

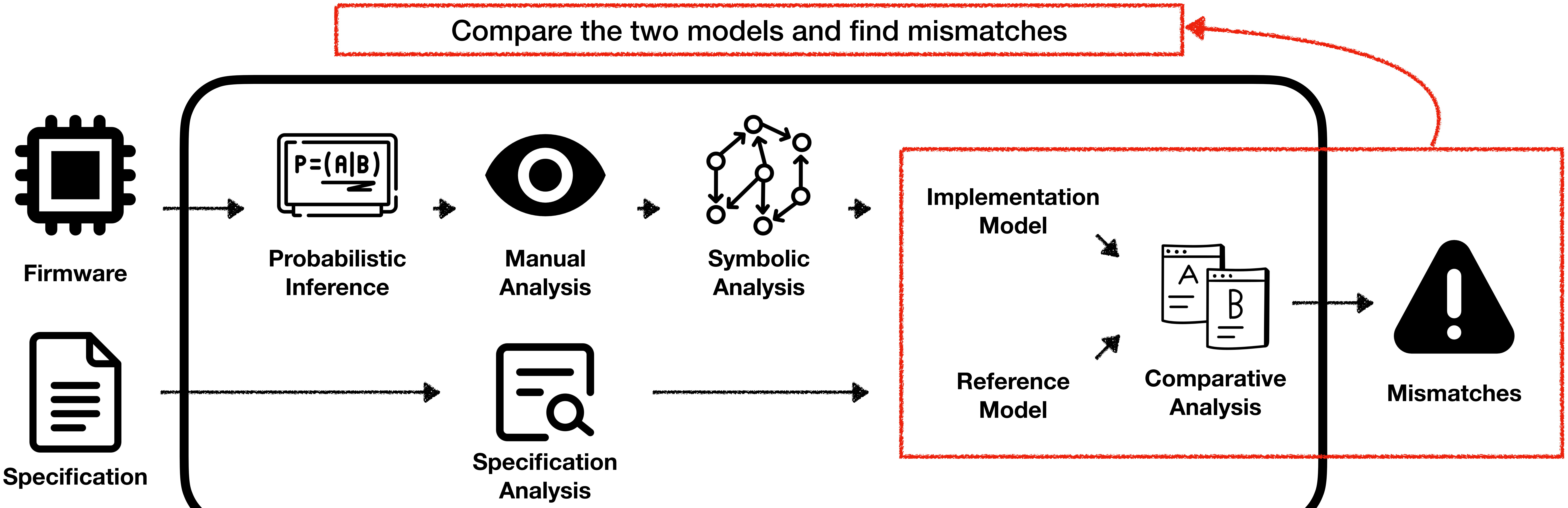
CheckSeq(message) == true && ValidateMac(message) == true

→ **Message is accepted!**

```
// 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 0x4B;
                case 0x4E;
                case 0x52;
                case 0x54;
                case 0x46;
                    return true;
                default:
                    return false;
            }
        }
    }
}
```

BaseComp

Comparative Analysis



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

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

<The rank of the integrity protection function for each firmware>

Evaluation

How effectively can BaseComp discover bugs?

- Summary
  - 34 Mismatches
  - 29 True Positives
  - 5 False Positives

	Samsung	MediaTek	srsRAN	Total
Mismatches	9	10	15	34
False Positives	1	3	1	5
True Positives	8	7	14	29

Evaluation

How effectively can BaseComp discover bugs?

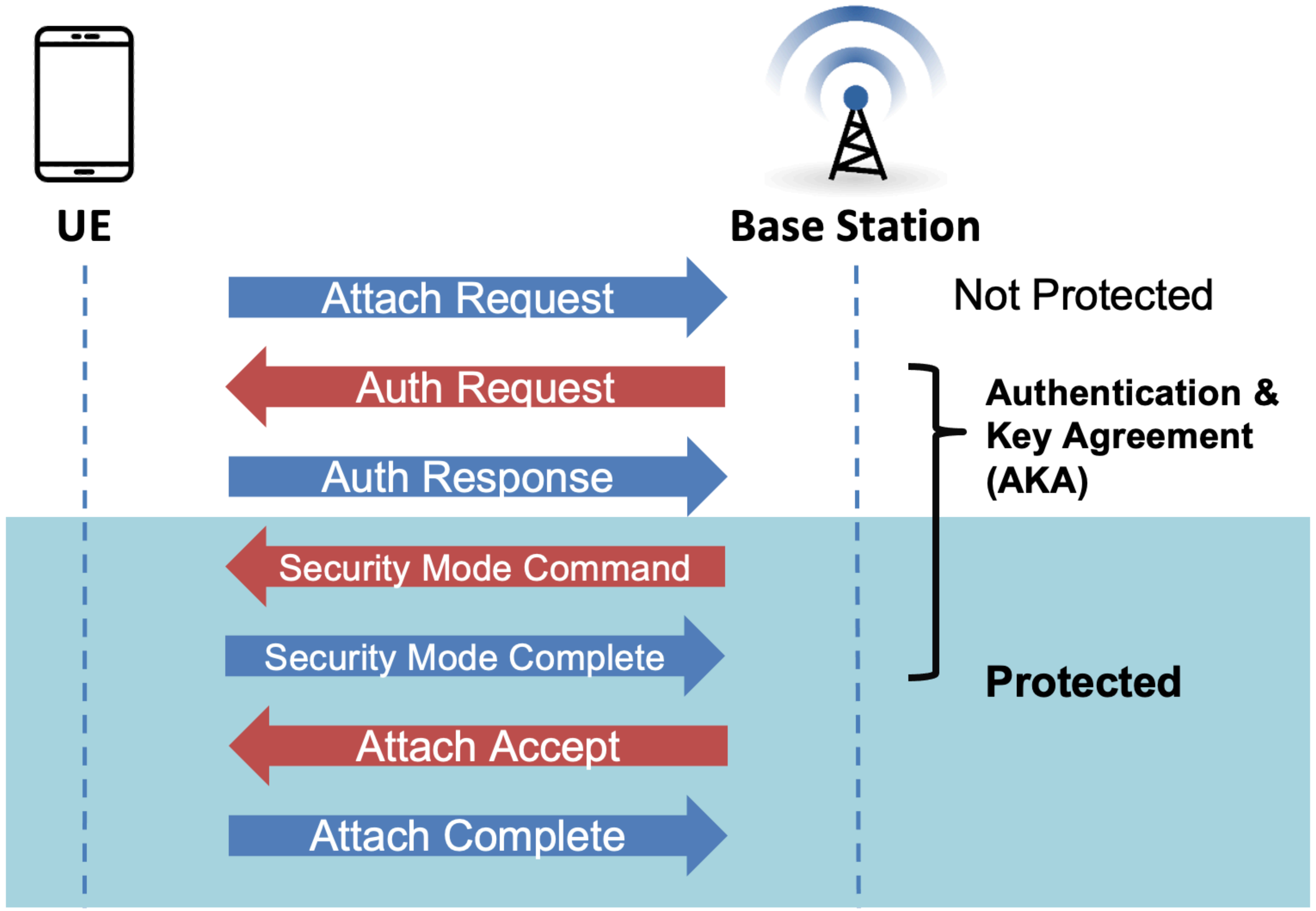
SECURITY State	Security Header Type	Message Type	Other Conditions	Mismatches in Samsung	Mismatches in MediaTek	Mismatches in srsRAN	Errors	Implication
INSECURE	3	Secure Mode Command		FP				
INSECURE	0 (Not Protected)	Identity Request	Identity Type ! = IMSI			FP		Info leak [29,43,48]
INSECURE	0 (Not Protected)	Attach Reject	EMM Cause == 25		FP	●	E1	DoS [48]
INSECURE	0 (Not Protected)	Tracking Area Update Reject	EMM Cause == 25		FP	●†	E2	DoS [12,13,65]
INSECURE	0 (Not Protected)	Service Reject	EMM Cause == 25		FP	●	E3	DoS [13]
INSECURE	! = 0, 1, 2, 3, 12	*		●			E4	Auth bypass
SECURE	0 (Not Protected)	Identity Request	Identity Type == IMSI	○	○	●	E5	Info leak [29,43,48]
SECURE	0 (Not Protected)	Authentication Request		○	○	●	E6	Location leak [29,31]
SECURE	0 (Not Protected)	Detach Accept		◑	◑	●†	E7	-
SECURE	0 (Not Protected)	Authentication Reject		◑	◑	●	E8	DoS
SECURE	0 (Not Protected)	Attach Reject	EMM Cause ! = 25	◑	◑	●	E9	DoS [54]
SECURE	0 (Not Protected)	Tracking Area Update Reject	EMM Cause ! = 25	○	◑	●†	E10	DoS [13,54,65]
SECURE	0 (Not Protected)	Service Reject	EMM Cause ! = 25	◑	○	●	E11	DoS [13,54]
*	0 (Not Protected)	Detach Request				●	E12	DoS [12,28,35]
*	0 (Not Protected)	EMM Information				●	E13	Info spoofing [35,48,49]
*	0 (Not Protected)	EMM Status				●	E14	- [35]
*	4	*				●	E15	Auth bypass
	Total number of		Mismatches	9	10	15		
			Bugs	8	7	14		

●: New bugs (neither bug nor its root cause previously reported), ◑: Duplicated bugs (not previously reported, but bugs with identical root causes were), ○: Old bug (previously reported) †: This bug has no implication due to the absence of handlers in the current implementation.

Case Study

NAS AKA Bypass Vulnerability

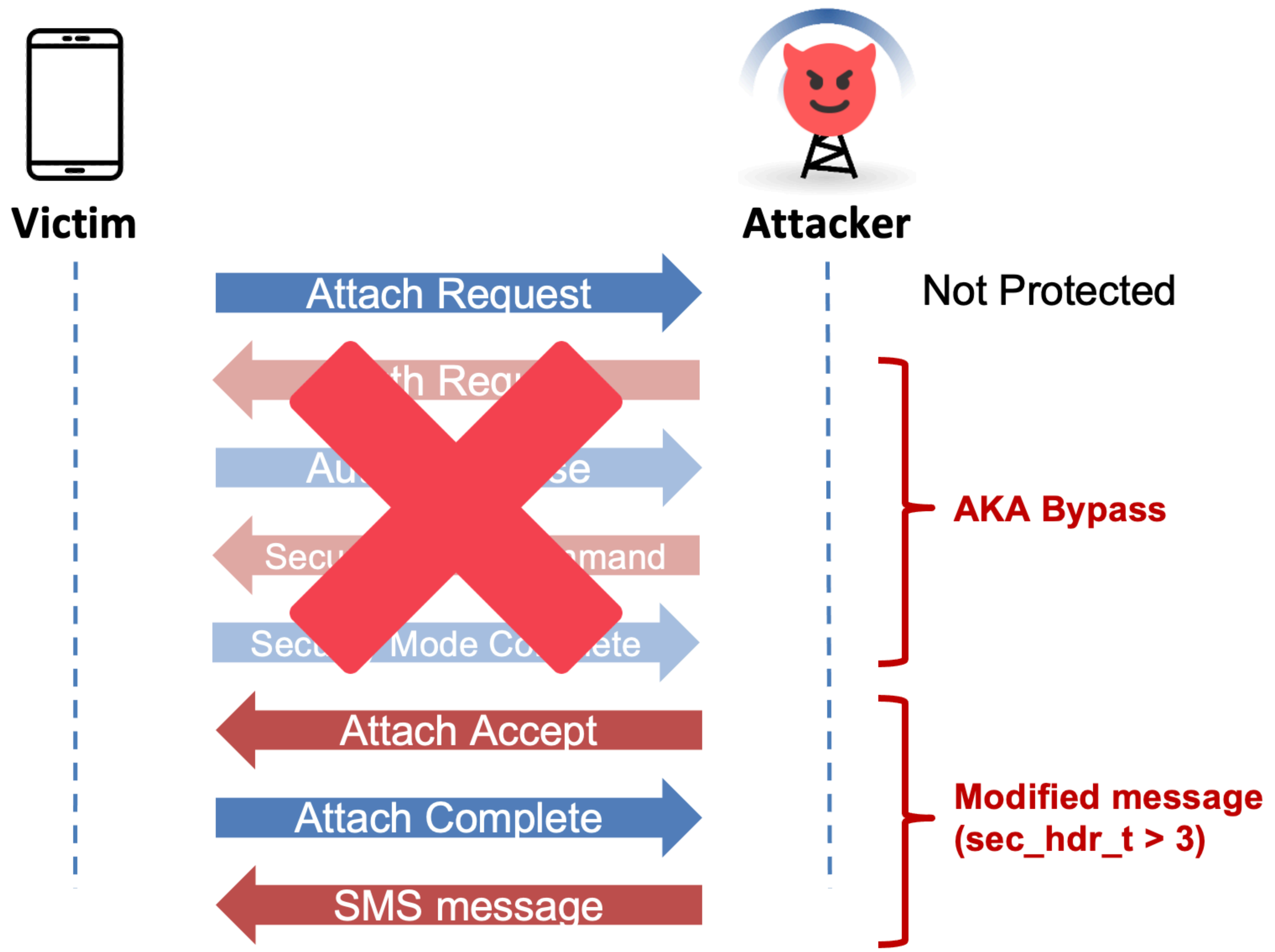
- NAS Authentication and Key Agreement



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
        - ...



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.

Good Questions

- Zhixian Jin
    - Author pointed out that challenges regarding to the full automation, but is it really possible to overcome these challenge to make full automation for any kind of software that try to find the bug?
- 오지오
    - Despite the risk of performance, are there any studies that fully automated these kinds of analyses in cellular softwares?
- 이승현
    - Is BaseComp still able to identify the integrity protection function properly if some or all of the MAC functions or message type comparing functions are inlined into another function?

Best Questions

- 윤태웅
  - Can you elaborate more on the trade-offs between static and dynamic techniques in the context of integrity protection analysis, and when might one approach be more suitable than the other?
- 오지오
  - Are there any ML approaches to convert these natural languages to structured data for software analyses or for attack?
- 오성룡
  - Is it hard for an attacker to find vulnerability on Hexagon architecture?

Backup Slides

Case Study

NAS AKA Bypass Vulnerability

- NAS Authentication and Key Agreement bypass

sec_state != SECURE

+

sec_hdr_type != 0

+

sec_hdr_type > 3

→ **Message is accepted!**

(regardless of any other element of the message)

```
// 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 IsProtected(void* message) {
    uint8_t sec_hdr_type = GetSecHdrType(message);
    return sec_hdr_type != 0 && sec_hdr_type <= 3;
}

bool CheckAllowableInNonSecure(void* message) {
    // Returns true if the 'message' is specified
    // as exceptions in TS 24.301.
    ...
}

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

    if (sec_state == SECURE) { ... }

    else { // INSECURE
        if (sec_hdr_type == 0)
            return CheckAllowableInNonSecure(message);
        else {
            // BUG: In the INSECURE state,
            // this function returns true
            // if sec_hdr_type is non-zero yet invalid.
            return true;
        }
    }
}
```

ML used in Cellular Security

- Sherlock on Specs
  - Utilizes natural language processing and machine learning on 3GPP specification to perform automated reasoning on events.
  - Yi Chen et al., "Sherlock on Specs: Building LTE Conformance Tests through Automated Reasoning", Usenix Security 2023