Artificial Intelligence & Cybersecurity

Formal Security Analysis of ZigBee Protocol

Student: Sebastiano Morson

5th August, 2025

What is ZigBee?

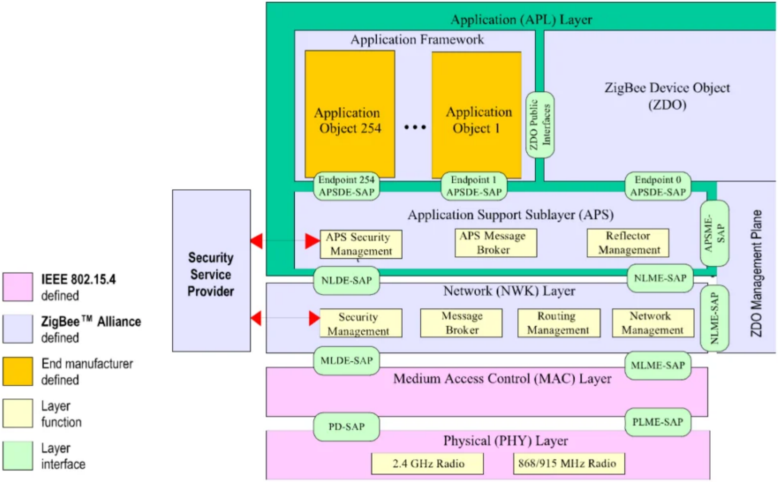
- Low-power, low-cost wireless protocol for IoT.
- Based on IEEE 802.15.4 for physical and MAC layers.
- Used in smart homes, industry, healthcare.

Bluetooth vs Wifi vs Zigbee

Wireless Parameter	**Bluetooth**	**Wi-Fi**	**ZigBee**
Frequency band	2.4 GHz	2.4 GHz	2.4 GHz
Physical/MAC layers	IEEE 802.15.1	IEEE 802.11b	IEEE 802.15.4
Range	9 m	75 to 90 m	Indoors: up to 30 m Outdoors (line of sight): up to 100 m
Current consumption	6 mA (Tx mode)	400 mA (Tx mode) 20 mA (Standby mode)	25–35 mA (Tx mode) 3 μA (Standby mode)
Raw data rate	1 Mbps	11 Mbps	250 Kbps
Protocol stack size	250 KB	1 MB	32 KB 4 KB (for limited function end devices)
Typical network join time	>3 sec	Variable, typically 1 sec	Typically 30 ms
Maximum number of nodes per network	7	32 per access point	64 K

Table 1: Comparison of Wireless Standards: Bluetooth, Wi-Fi, and ZigBee

Zigbee Protocol Stack



Main Device Types:

- **ZigBee Coordinator (ZC):** Central controller that initializes and manages the network. Assigns addresses and keeps track of topology and device status. Only one per network.
- **ZigBee Router (ZR):** Intermediate nodes that extend coverage and route packets. Communicate with ZC, other ZRs, and End Devices.
- **ZigBee End Device (ZED):** Battery-powered, low-energy devices. Communicate only with a parent (ZC or ZR). Do not route traffic.

Example of a Real ZigBee Network

- **Coordinator (ZC):** Smart hub (e.g., Amazon Echo Plus, SmartThings) that manages the network.
- **Routers (ZR):** Smart bulbs distributed across rooms, relaying messages.
- **End Devices (ZED):** Battery-powered sensors (e.g., temperature/humidity) that send data periodically.

ZigBee Security Goals

- **Authentication:** Ensure entity identity (Lowe hierarchy).
- **Confidentiality:** Avoid data disclosure $\rightarrow$ AES-128.
- **Integrity:** Detect tampering $\rightarrow$ MIC, Frame Counter.
- **Main threats:** eavesdropping, replay, key compromise.

- Paper focuses on four key security procedures in ZigBee:
  - Network Key Sharing
  - Joining a Secure Network
  - Application Key Establishment
  - Network Key Update
- Comparison of ZigBee 1.0 vs 3.0 implementation for each process

Goal: Distribute the network key securely during the joining phase.

- **ZigBee 1.0:** Uses pre-configured global key shared with all devices
  - If compromised, attacker can decrypt network key
- **ZigBee 3.0:** Uses installation code + EUI64
  - Processed using AES-MMO hash to derive a unique pre-shared key
  - Installation code shared out-of-band with Trust Center

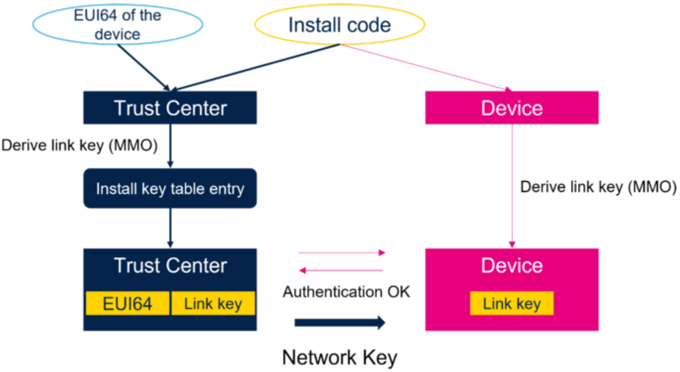


Figure: Derivation of the pre-configured key from install code using AES-MMO hash

Joining a Secure Network

- Devices send **Beacon Request** to find parent
- Upon MAC association, Trust Center is notified
- Trust Center authenticates device and sends Network Key encrypted with Link Key
- Post-joining: device must request **Trust Center Link Key update**
- Ensures secure communication from the start

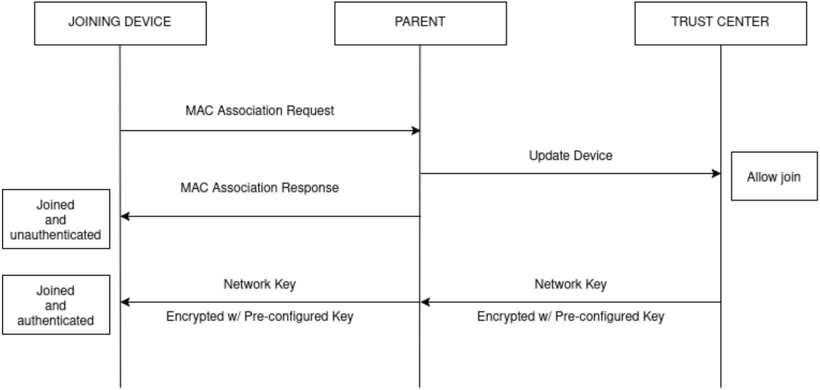


Figure: Joining a secured network

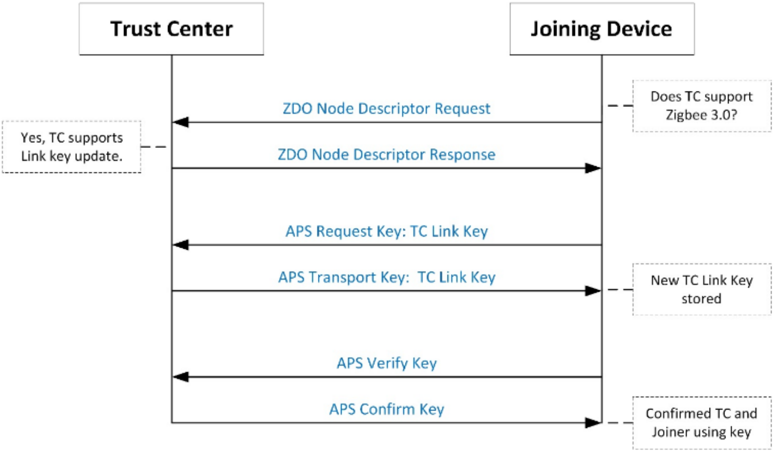


Figure: Updating of the Trust Center Link Key

Application Key Establishment

Goal: Enable secure end-to-end communication

- Step 1: Initiator requests Application Link Key to Coordinator
- Step 2: Coordinator replies with **Transport Key** message
- Step 3: Coordinator sends same key to recipient device
- Result: Unique Application Key shared between the two devices

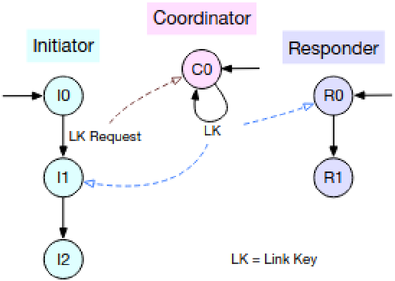


Figure: Application Key Establishment

Network Key Update

- Prevent replay attacks via **Frame Counter** (max 0xFFFFFFFF)
- When value > 0x80000000 ⇒ initiate key update
- Distribution modes:
  - **Broadcast:** encrypted with old Network Key
  - **Unicast:** encrypted with Link Key per device
- Final step: Key Switch command issued to activate new key
- If a device misses update ⇒ forced to rejoin network

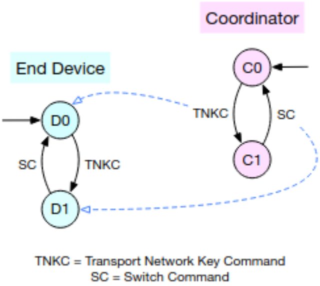


Figure: Network Key Update

- ZigBee 1.0: uses global preconfigured key (vulnerable).
- ZigBee 3.0: introduces install-code $\Rightarrow$ unique key per device.
- Secure key distribution via out-of-band.
- Improved joining, authentication, and key update mechanisms.

Introduction to Tamarin Prover

- Tool used to formally verify the specifications
- Symbolic model checker for cryptographic protocol analysis.
- Inputs: protocol model + security properties.
- Output: proof of correctness or counterexample trace.
- Verifies properties like: secrecy, authentication, integrity.

- Protocol behavior modeled using **rewriting rules**.
- Rules define a **labeled transition system** (LTS).
- Basic components:
  - **Terms:** bitstrings, variables, messages.
  - **Facts:** represent info state or actions.
  - **State:** multiset of facts at a given point.
- Adversary capabilities also modeled with rules.

Terms and Facts in Tamarin

Terms: represent data used in the protocol

- **Constants:** e.g., `alice`, `bob`, `k1`
- **Variables:** e.g., `x`, `y`, `k`
- **Functions:** e.g., `enc(m,k)`, `hash(m)`

Facts: building blocks of the system state

- **State info:** e.g., `Key(alice,k1)`
- **Events:** e.g., `ReceivedMessage(m1)`
- **Input control:** e.g., `In(m1)`
- Temporary facts: consumed when rules apply
- Persistent facts: stay across rule applications (prefix !)
- Built-ins: `In(x)`, `Out(x)`, `K(x)`, `Fr(x)`

Traces and Lemmas in Tamarin

Trace: sequence of rule applications (events)

- Each trace represents a possible protocol execution
- Events carry labels (e.g., `Send(m)`, `Receive(m)`)
- Tamarin verifies security properties over all possible traces

Lemmas: properties to prove

- **Exists-trace:** show one valid trace satisfies formula
- **All-traces:** prove no trace violates formula (i.e., no counterexample found)
- Use first-order logic with atoms like:
  $\texttt{F@}i$, $\texttt{i} < \texttt{j}$, $\texttt{t}_1 \approx \texttt{t}_2$, $\neg$, $\wedge$, $\exists$, $\forall$, $\perp$

Rewriting Rules and Transitions

- Rule syntax: `L -[A]-> R`
- `L`: **preconditions** (facts), `A`: **actions** (labels), `R`: **conclusions** (facts)
- A set of rewriting rules + current state defines an LTS
- Rule applies if a ground instance of `L` matches the current state

$$S_{t+1} = S_t \setminus^{\#} L \cup^{\#} R$$
$$T_{t+1} = \langle a_0, \ldots, a_{t-1}, a_t \rangle$$

- Built-in rules:
  - `Out(x) -[]-> K(x)`: adversary learns x
  - `K(x) -[K(x)]-> In(x)`: inject message
  - $\emptyset$ `-[]-> Fr(x)`: generate fresh

Equational Theories in Tamarin

- Tamarin supports **convergent equational theories with the finite variant property**
- set of algebraic rules to state when two terms are considered equivalent under a given equational reasoning system
- Used to model cryptographic operations (e.g., encryption, hashing)
- Important properties:
  - **Confluence:** rewriting leads to a unique normal form
  - **Termination:** guarantees rewriting ends
  - **Finite Variant:** only a finite set of normal forms for each term
- User-defined theories must satisfy these properties manually

Violating Equational Theory Properties

- **Equational theories** must be well-behaved to ensure sound security protocol analysis.
- Violating these properties can lead to unexpected behaviors and incorrect results in a model.

Example Violations

- **Confluence (Non-Unique Normal Forms):**
  - **Rule 1:** decrypt(key, crypt(key, m))=m
  - **Rule 2:** decrypt(pk(key), crypt(key, m))=m

- **Violation:** Starting with decrypt(pk(key), crypt(key, m)),
  - Rule 2 applies directly, giving the unique normal form m.
  - Rule 1 does not apply, leaving the term unchanged.
- **Result:** Depending on the order in which rules are applied, we get different outcomes. This demonstrates a non-unique normal form.

Termination: The Problem of Infinite Loops

- **Violation:** Consider these two rules that rewrite each other.
  - **Rule 1:** f(g(x))=h(x)
  - **Rule 2:** h(x)=f(g(x))
- **Result:** A rewrite loop is created. Starting with f(g(a)), we can rewrite to h(a) (by Rule 1) and then back to f(g(a)) (by Rule 2) infinitely. This prevents the rewriting from ever terminating.

Finite Variant: The Problem of Endless Forms

- **Violation:** A rule that generates infinite, distinct normal forms.
  - **Rule:** encrypt(k, m)=encrypt(fresh(k), m)
- **Result:** The fresh function generates a new, unique key each time it is called. Rewriting encrypt(k, m) can lead to an infinite set of equivalent but distinct normal forms: encrypt(fresh(k), m), encrypt(fresh(fresh(k)), m), etc.

Given a trace T and a valuation Θ:

$$
\begin{aligned}
T, \Theta \vDash F@i &\iff 1 \leq \Theta(i) \leq n \wedge \Theta(F) \in T[\Theta(i)] \\
T, \Theta \vDash i \dot{<} j &\iff \Theta(i) < \Theta(j) \\
T, \Theta \vDash i \doteq j &\iff \Theta(i) = \Theta(j) \\
T, \Theta \vDash t_1 \approx t_2 &\iff \Theta(i) \approx \Theta(j) \\
T, \Theta \vDash \neg\varphi &\iff T, \Theta \nvDash \varphi \\
T, \Theta \vDash \varphi \wedge \psi &\iff T, \Theta \vDash \varphi \wedge T, \Theta \vDash \psi \\
T, \Theta \vDash \exists x : s.\varphi &\iff \exists v \in D_s : T, \Theta[x \rightarrow v] \vDash \varphi
\end{aligned}
$$

A protocol P satisfies a lemma ϕ iff:

$$P \vDash \phi \iff trace(\phi) \subseteq trace(P)$$

- Protocol split into: key generation, joining, key updates
- Different rules for ZigBee 1.0 and 3.0

Pre-configured Key Rule:

```
rule D_pck_generation:A
  [ Fr(~pck) ]
  --[ SecretPCK(~pck) ]->
  [ !PCK($D,$C,~pck) ]
```

- Fr(pck): generates fresh secret key
- SecretPCK: labels it for secrecy analysis
- !PCK(...): persists the key in system state

Key Revelation for Lemmas

```
rule Reveal_nk:
  [ !NwkKey(C,~nk) ] --[ RevNK(C) ]-> [ Out(~nk) ]
```

- Simulates attacker learning the network key
- Used to prove secrecy lemmas by contradiction

New Node Joining: Secure Channels

```
rule ChanOut_S:
  [ Out_S($A,$B,x) ]
  --[ ChanOut_S($A,$B,x) ]->
  [ !Sec($A,$B,x) ]

rule ChanIn_S:
  [ !Sec($A,$B,x) ]
  --[ ChanIn_S($A,$B,x) ]->
  [ In_S($A,$B,x) ]
```

- Defines secure send/receive behavior
- !Sec(...): proves the message was securely exchanged

New Node Joining: Beacon Request

```
rule D1_Beacon_req:
  [ !PCK($D,$C,~pck) ]
  --[ Beacon_req() ]->
  [ Out(<$D,'0x07'>), BeaconReq($D), !ZigbeeV3(), !ZigbeeV1() ]
```

- Beacon request modeled for both ZigBee 1.0 and 3.0
- Next rule depends on protocol version

```
rule D2_3_association_req:
  [ ..., !ZigbeeV3() ] -> [ ..., Out_S(D,C,~pck) ]

rule D2_1_association_req:
  [ ..., !ZigbeeV1() ] -> [ ..., Out(<D,'pck'>) ]
```

- 3.0: key sent over secure channel
- 1.0: key exposed to attacker (insecure channel)

Trust Center Link Key Update

```
rule D4_NTLK_verify:
  [ In(senc(ntlk, ~pck)), ..., !PCK(D,C,~pck) ] -> [...]

rule C4_NTLK_verified:
  [ In(senc(...)), ... ] -> [ Counter(D,'0'), !SendMsg(D,C) ]
```

- `senc(ntlk,  pck)`: ciphertext modeled as input
- Decryption applied via `sdec(...)`
- Rule ensures correctness of key verification

Network Key Update Trigger

```
rule Frame_counter_increase:
  [ !SendMsg(D,C), Counter(D, val) ]
  --[ Msg_Send() ]->
  [ Counter(D, inc(val)), Out(inc(val)) ]
```

- Counter incremented on every message
- Counter is leaked `Out(...)`: represents protocol behavior
- Used to simulate threshold-triggered key update

Example: joining implies verification

$\exists$-trace $(\exists \#i.\ \text{Beacon\_req}()@\#i \Rightarrow \exists y \#j.\ \text{NTLK\_verified}(y)@\#j)$

- Guarantees correctness of session key validation
- Similar structure used for other protocol phase checks

General form:

$$\forall\text{-trace}(\forall x \# i.\ \mathrm{SecretNK}(x)@\#i \Rightarrow \neg(\exists \# j.\ \mathrm{K}(x)@\#j) \mid \ldots)$$

- Ensures a key remains secret unless explicitly revealed
- Exceptions: `RevNK`, `RevPCK`, etc.
- Same scheme for `NTLK`, `PCK`, `LK`

Authentication: Key Agreement

```
lemma key_agreement:
 "All nk1 nk2 coordinator device #i #j.
  Send_Network_Key(nk1,coordinator,device) @ i
  & Send_Network_Key(nk2,coordinator,device) @ j
    ==> nk1 = nk2"
```

- Prevents mismatches in session keys
- Same device must always receive identical key

The Problem with the key_agreement Lemma

Original Lemma

```
lemma key_agreement:
 "All nk1 nk2 coordinator device #i #j.
  Send_Network_Key(nk1,coordinator,device) @ i
  & Send_Network_Key(nk2,coordinator,device) @ j
    ==> nk1 = nk2"
```

- **Observation:** This lemma states that for a given device, any two network keys sent by the coordinator must be identical.
- **Problem:** This property **prevents key updates**. If the coordinator sends a new key (nk2) to the device, the lemma is violated, as nk1 $\neq nk2$.

- The lemma is too strong; it guarantees a key's immutability, not its secure delivery.

The Corrected Lemma for Key Updates

Corrected Lemma

```
lemma key_update_agreement:
 "All nk1 nk2 coordinator device version1 version2 #i #j.
  Send_Network_Key(nk1,coordinator,device,version1) @ i
  & Send_Network_Key(nk2,coordinator,device,version2) @ j
  & version1 = version2
    ==> nk1 = nk2"
```

- **Improvement:** We add a version parameter to the Send_Network_Key predicate.
- **Logic:** The lemma now only enforces that if two keys are sent with the **same version**, they must be identical.
- **Result:** This allows the coordinator to send a new key with an updated version (version2 != version1) without violating the lemma. It guarantees that the key is consistent **within a specific version**, enabling secure key updates.

Authentication: Key Uniqueness

```
lemma uniqueness_of_key:
 "All key coordinator device #i #j.
  Send_Network_Key(key,coordinator,device) @ i
  & Send_Network_Key(key,coordinator,device) @ j
    ==> #i = #j"
```

- Prevents multiple sends of same key at different times
- Useful against replay attacks

Authentication: Weak Agreement

```
lemma weak_agreement:
  " All c d x1 #i.
    NwkKeyRecv(d,c,x1) @ #i
    ==>
     ((Ex x2 #j. Send_Network_Key(c,d,x2) @ #j)
    |(Ex #r. RevNK(c) @ r)
    | (Ex #r. RevPCK(d,c) @ r))"
```

- Weakest level of authentication
- A device can only claim it received a key if:
  - The coordinator actually sent a key to it, or
  - The key was compromised (`RevNK`), or
  - The pre-configured key (PCK) was leaked (`RevPCK`)
- Prevents unjustified claims of key reception

```
lemma non_injective_agreement:
  " All c d x #i.
    NwkKeyRecv(d,c,x) @ #i
    ==>
     ((Ex #j. Send_Network_Key(c,d,x) @ #j)
    |(Ex #r. RevNK(c) @ r)
    | (Ex #r. RevPCK(d,c) @ r))"
```

- Same key must be the one received
- Allows reuse in multiple sessions

Authentication: Injective Agreement

```
lemma injective_agreement:
  " All c d x #i.
    NwkKeyRecv(d,c,x) @ #i
    ==>
     (Ex #j. Send_Network_Key(c,d,x) @ #j
     & j < i)
    |(Ex #r. RevNK(c) @ r)
    | (Ex #r. RevPCK(d,c) @ r)"
```

- Strongest level of authentication
- Ensures key **received** was **sent previously** by the coordinator
- Enforces **temporal order**: $j < i$
- Prevents **replay attacks** or reuse of the same key in multiple sessions

- **ZigBee 3.0**: All security properties verified
- **ZigBee 1.0**: Vulnerability in network key secrecy detected
- Tamarin highlights violation of `secrecy_NK` lemma
- Violation traced to use of pre-configured keys without secrecy requirement

```
lemma secrecy_NK:
  all-traces
  "∀ x #i.
          (SecretNK( x ) @ #i) ⇒
          ((((¬(∃ #j. K( x ) @ #j)) ∨ (∃ C #r. RevNK( C ) @ #r)) ∨
            (∃ C D #r. RevPCK( D, C ) @ #r)) ∨
           (∃ C D #r. RevNTLK( C, D ) @ #r))"
simplify
solve( !KU( ~nk ) @ #vk )
  case C2_1_Send_Nwk_key
  SOLVED // trace found
next
  case C2_3_Send_Nwk_key
  by sorry
next
  case Reveal_nk
  by contradiction /* from formulas */
qed
```

Figure: Tamarin found a trace that violates the lemma `Secrecy_NK`

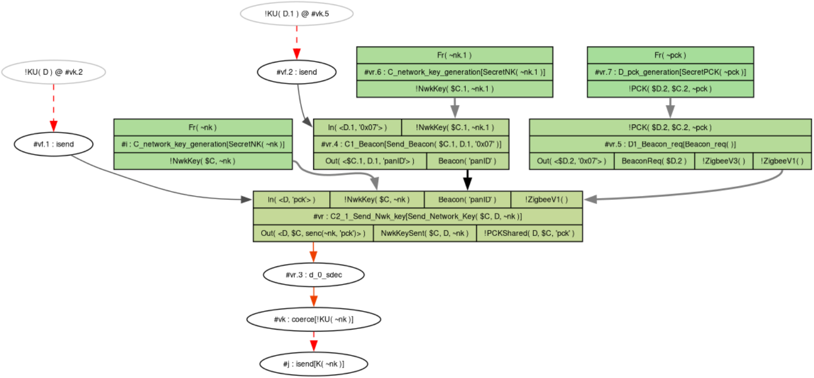


Figure: Network Key Disclosure trace showing attacker decrypts the last message

Conclusions

- ZigBee 1.0 fails to guarantee key secrecy under realistic conditions
- ZigBee 3.0 addresses vulnerabilities with updated key handling mechanisms
- Tamarin Prover proves effective for:
  - Finding subtle protocol flaws
  - Validating protocol correctness through formal methods