Breaking Pacemaker Authentication with Formal Methods

Christian Coduri
christian.coduri@polito.it

whoami

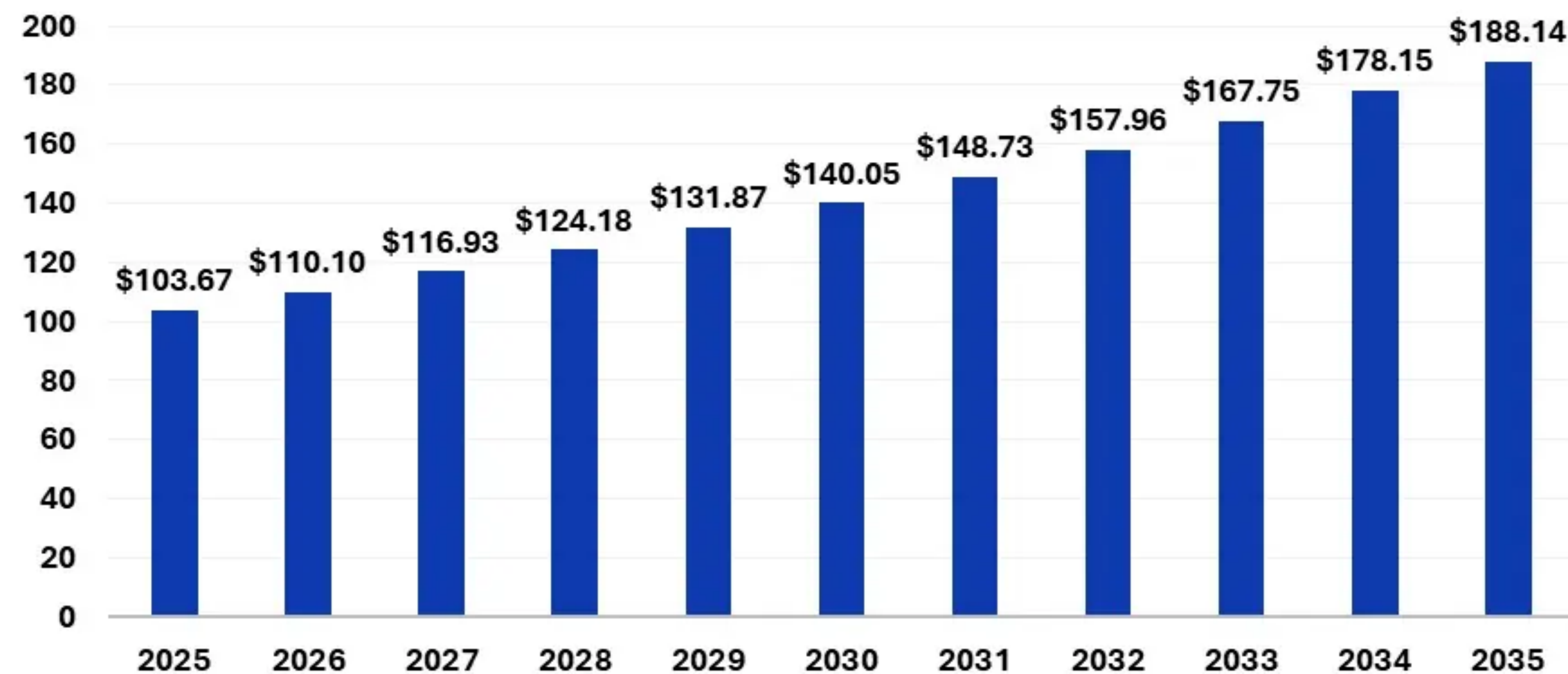
- PhD Student, Computer and Control Engineering @ **PoliTO**
- MSc in Cybersecurity Engineering @ **PoliTO**
- BSc in Computer Systems and Network Security @ **UniMI**
- **Research**: AI Security… *but not today*

Agenda

1. **IMD Ecosystem and Threat Landscape**
2. **Authentication and Access Control** in IMDs
3. **Formal Verification** for Security Protocols
4. **Case Study**: Formal Verification of IMDGuard
    - System Overview
    - Modeling in ProVerif
    - Verification Results
    - Identified Attacks and Vulnerabilities

Motivation

- Implantable Medical Devices are **life-critical** and increasingly connected.
- Attacks can compromise **safety, privacy, or device functionality**.
- Existing security protocols often **lack rigorous formal validation**.
- The global implantable medical devices market is estimated at USD 103.7 billion in 2025 and projected to grow to **USD 110.1 billion in 2026**.



Statistics from https://www.precedenceresearch.com/implantable-medical-devices-market

(Real) Motivation

Implantable Medical Devices

IMDs are devices that are surgically placed inside the human body and are designed to remain in place for extended periods.

Main purposes:

- Monitor critical physiological parameters
- Deliver therapeutic interventions

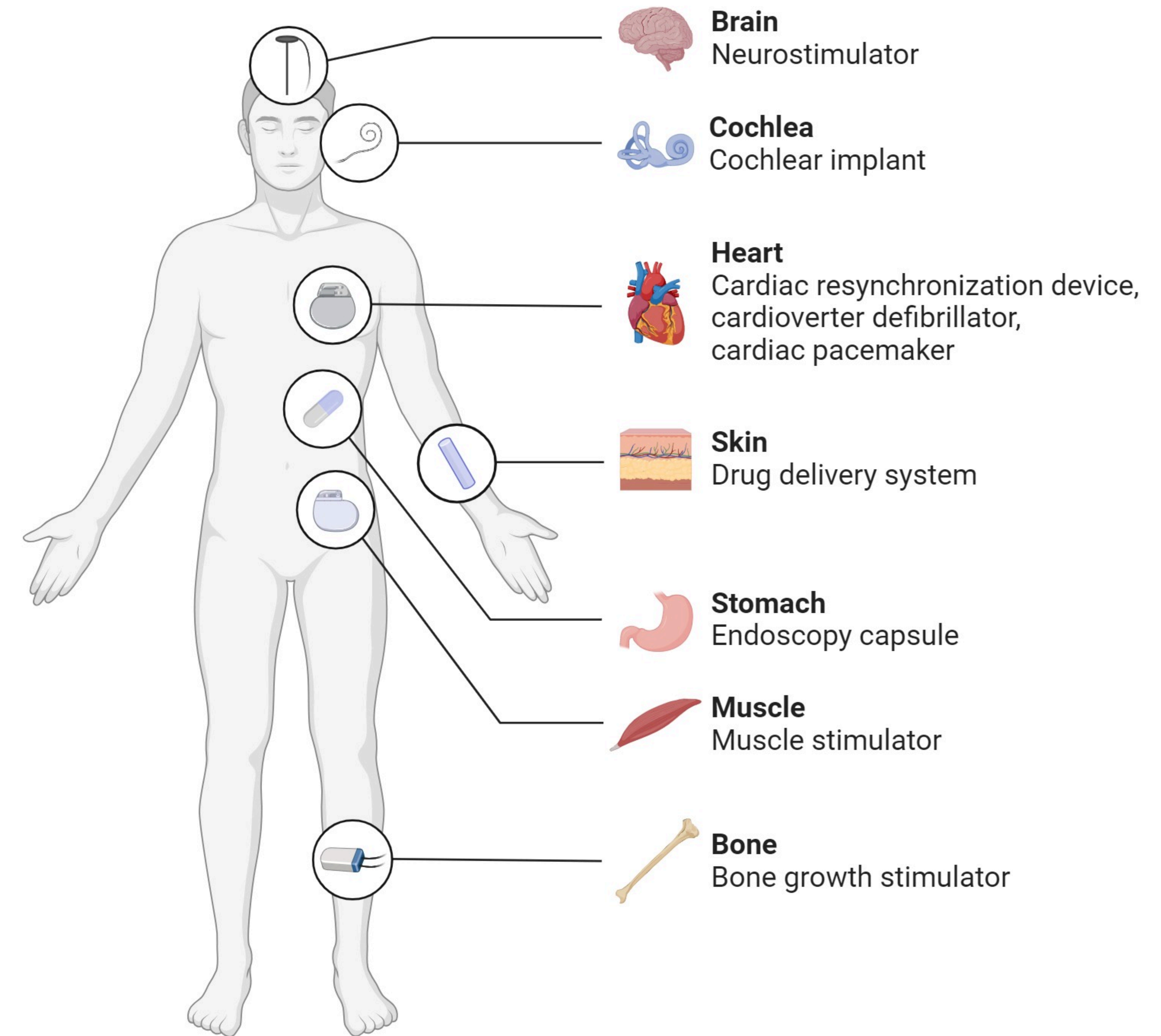


Image from https://www.biorender.com/template/implantable-medical-devices-locations

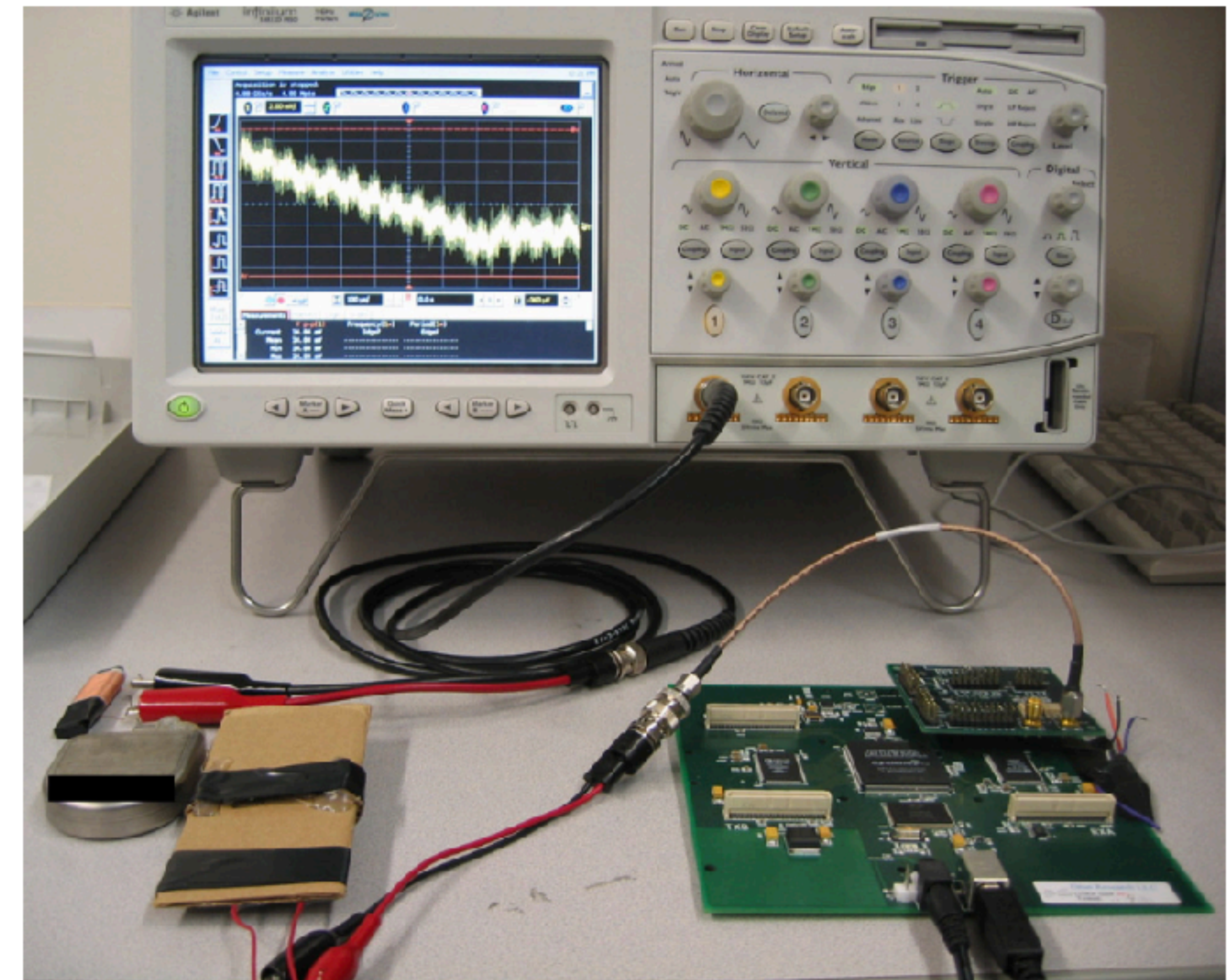
IMD Ecosystem

Tier-1
Tier-2
Tier-3
Pacemaker
Programmer
Wireless Channel
Monitor
Internet
Server

Zheng, Guanglou, et al. "Ideas and challenges for securing wireless implantable medical devices: A review." IEEE Sensors Journal 17.3 (2016): 562-576.

Tier-1 Attacks

An attacker can communicate directly with the IMD using **software-defined radio tools**.



Security Impact:

- **Privacy:** exposure of patient/device data
- **Integrity:** modification of stored data or therapy settings
- **Availability:** reduced battery life
- **Safety:** potential physical consequences from therapy-related attacks

Halperin, Daniel, et al. "Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses." 2008 IEEE Symposium on Security and Privacy (sp 2008). IEEE, 2008.

Tier-2 Attacks

Programmers and home monitors are **easier to access and compromise** than IMDs.

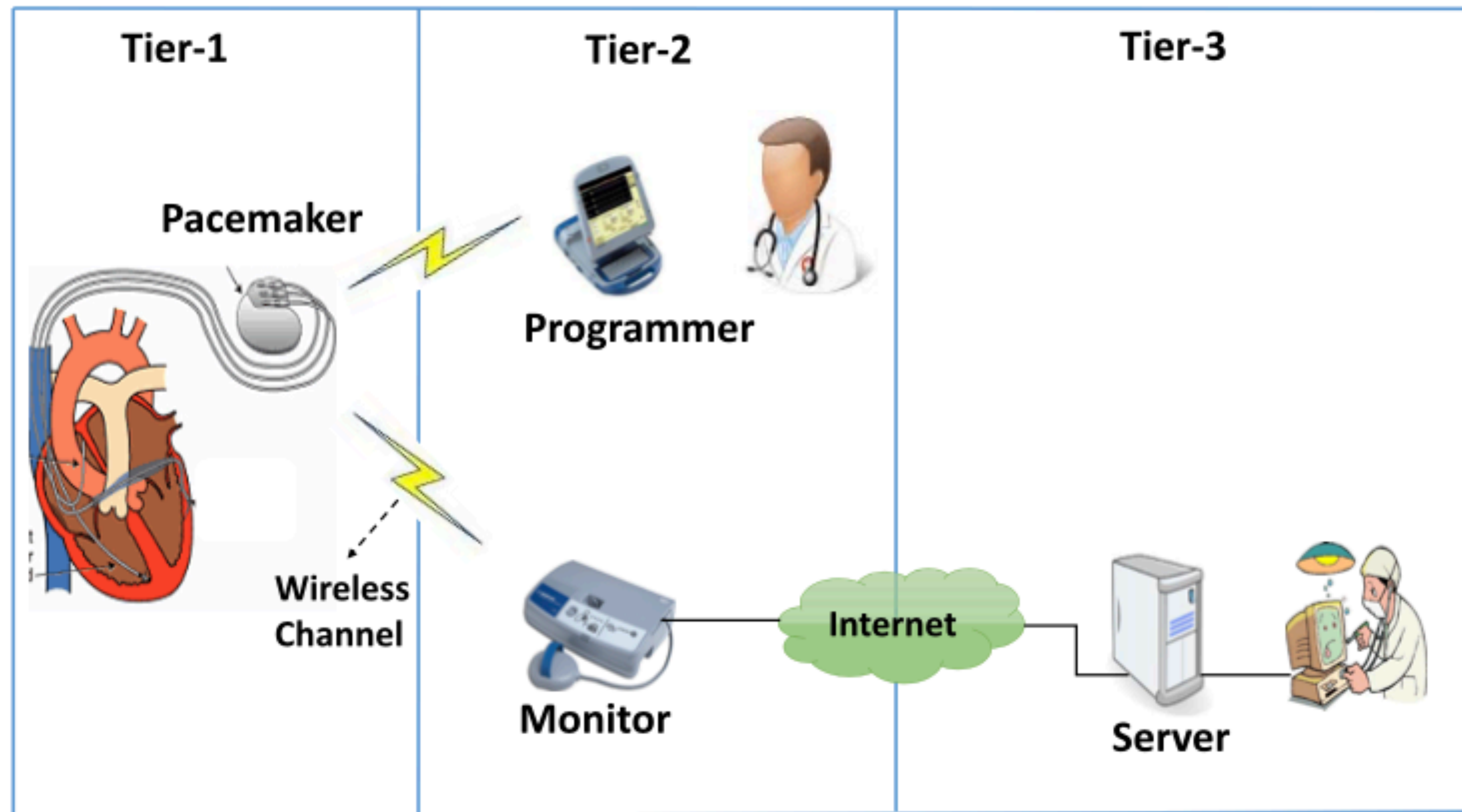
Main attack surfaces:

- Weak **firmware** protection
- Exposed **physical interfaces**
- Hardcoded **credentials** and exposed **patient data**
- Insufficient programming and RF communication **controls**

Image from: Sridharan, A., Farmer, D.M., Homoud, M. (2022). Pacemaker Interrogation and Programming. In: Hendel, R.C., Kimmelstiel, C. (eds) Cardiology Procedures. Springer, Cham.
Rios, Billy, and Jonathan Butts. "Security evaluation of the implantable cardiac device ecosystem architecture and implementation interdependencies." *WhiteScope, sl* (2017).

IMD Ecosystem

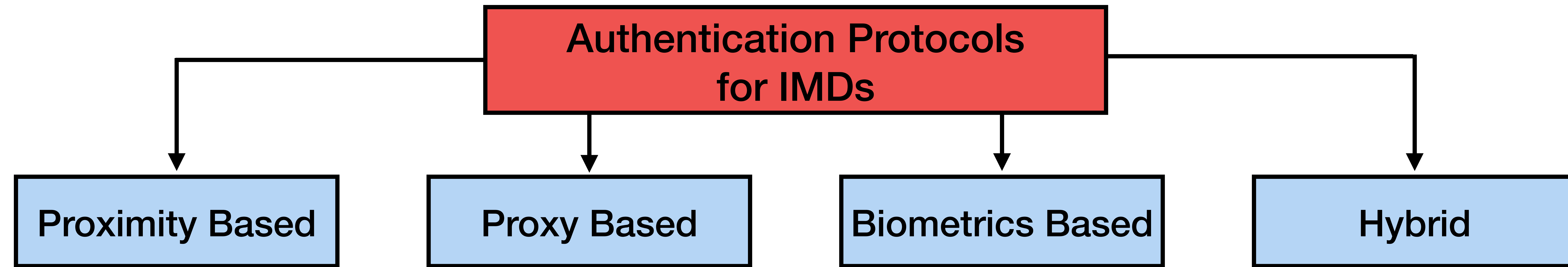
! IMD security depends on the whole system, not only on the implanted device.



IMD Ecosystem

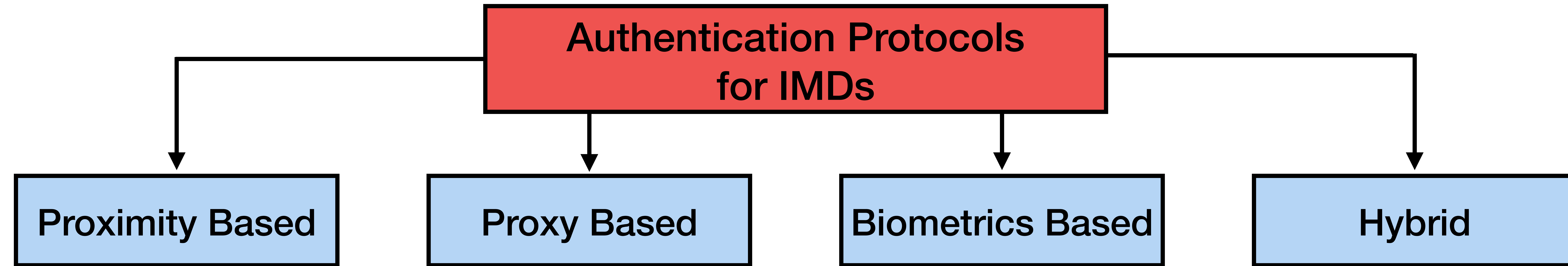
> ! <u>Observation</u>: Tier-1 ↔ Tier-2 wireless channel is a critical security boundary
>
> <u>Idea</u>: Security depends on **authentication and access control** on this channel

Authentication and Access Control



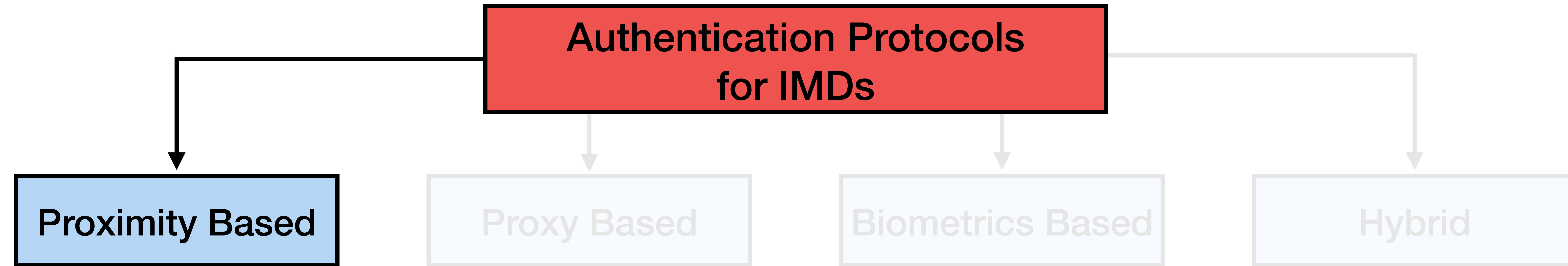
Ellouze, Nourhene, et al. "Security of implantable medical devices: Limits, requirements, and proposals." Security and Communication Networks 7.12 (2014): 2475-2491.

Authentication and Access Control



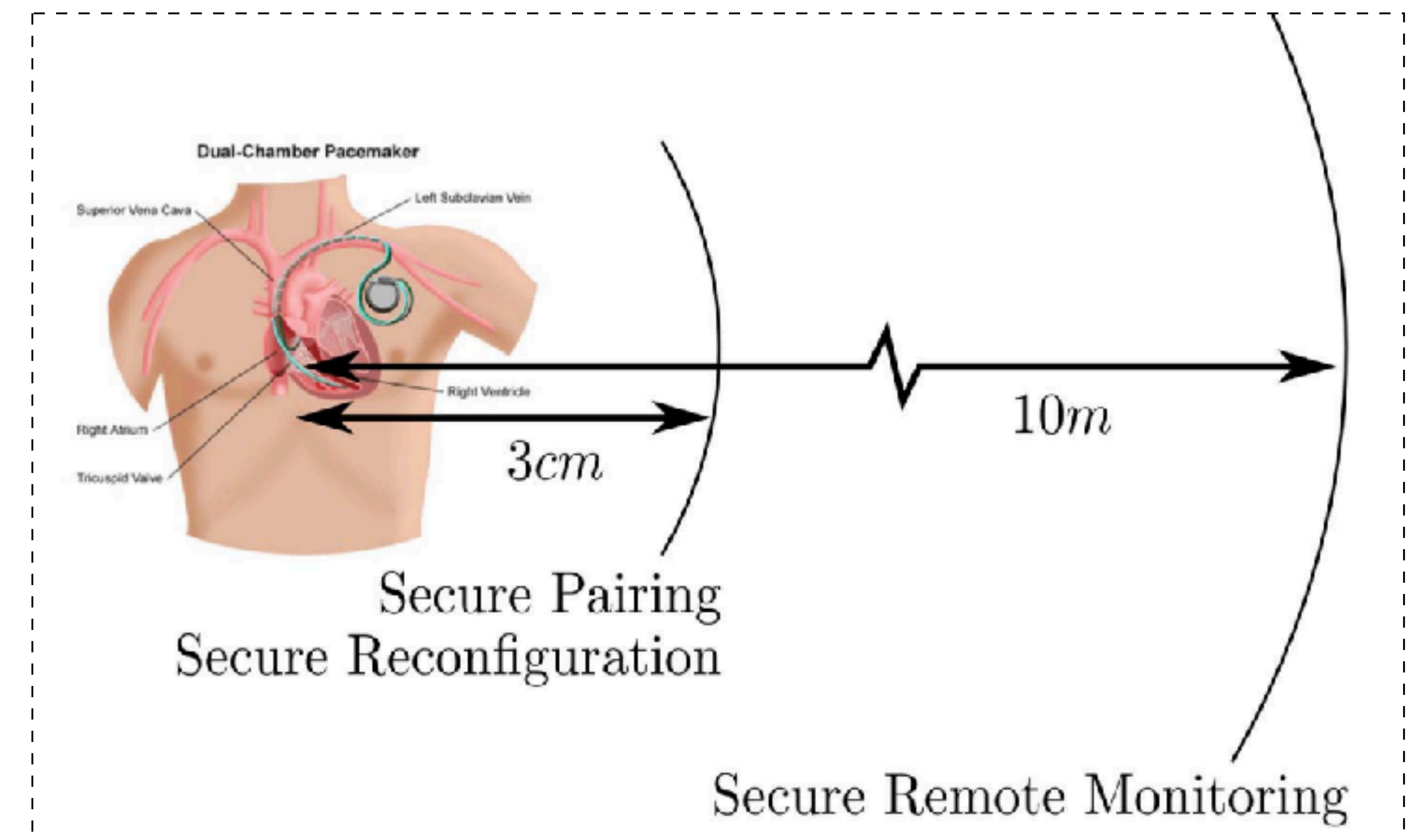
Ellouze, Nourhene, et al. "Security of implantable medical devices: Limits, requirements, and proposals." Security and Communication Networks 7.12 (2014): 2475-2491.

Authentication and Access Control



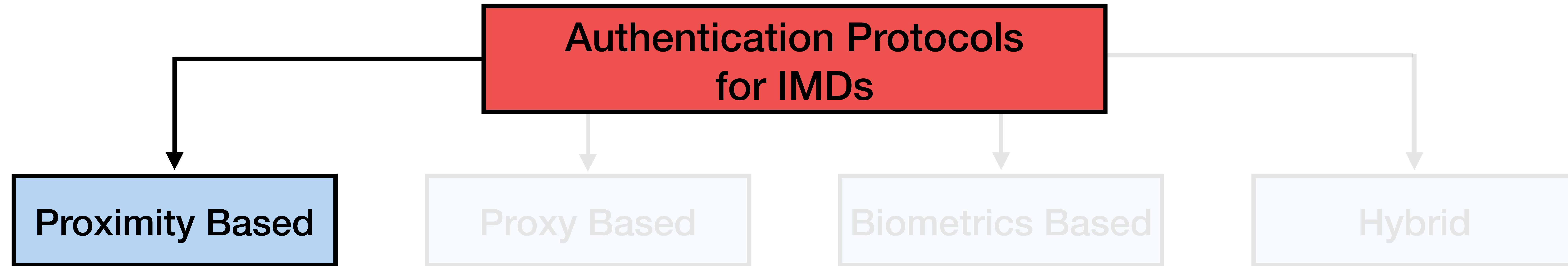
How They Work:

- Access is granted when the external device is within a **limited distance** from the IMD
- Common techniques: NFC, Ultrasound



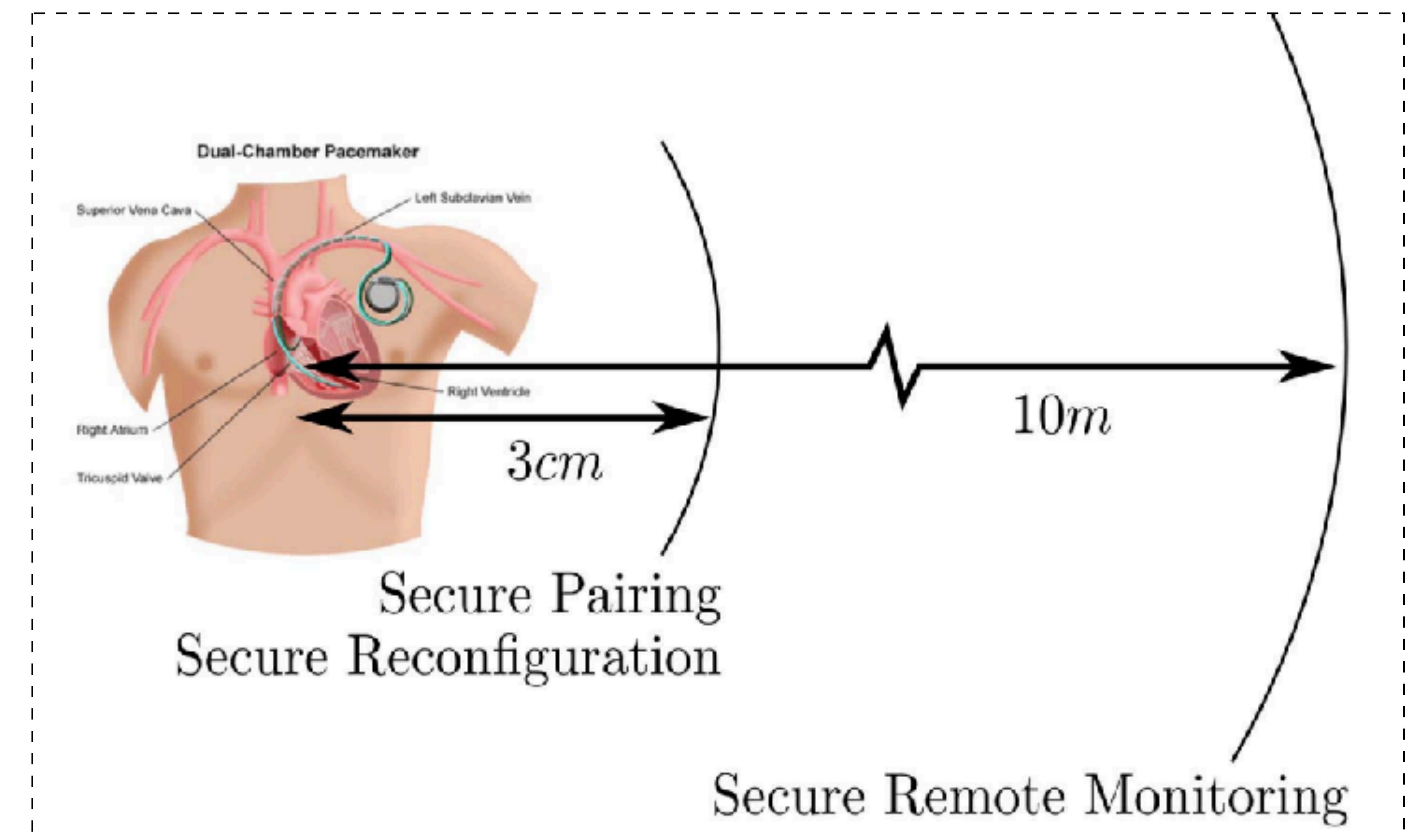
Zheng, Guanglou, et al. "Ideas and challenges for securing wireless implantable medical devices: A review." IEEE Sensors Journal 17.3 (2016): 562-576.

Authentication and Access Control



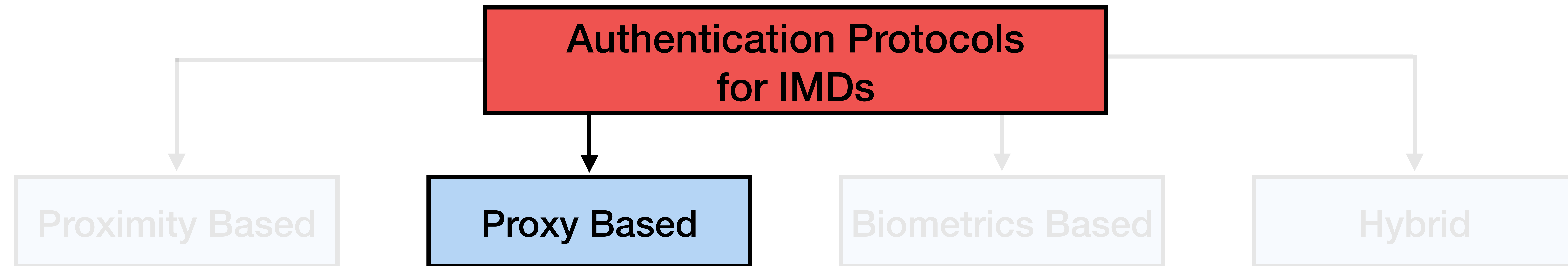
Assumptions:

- Physical **proximity** implies **authorization**
- Distance can be measured reliably
- Attackers cannot relay or amplify signals



Zheng, Guanglou, et al. "Ideas and challenges for securing wireless implantable medical devices: A review." IEEE Sensors Journal 17.3 (2016): 562-576.

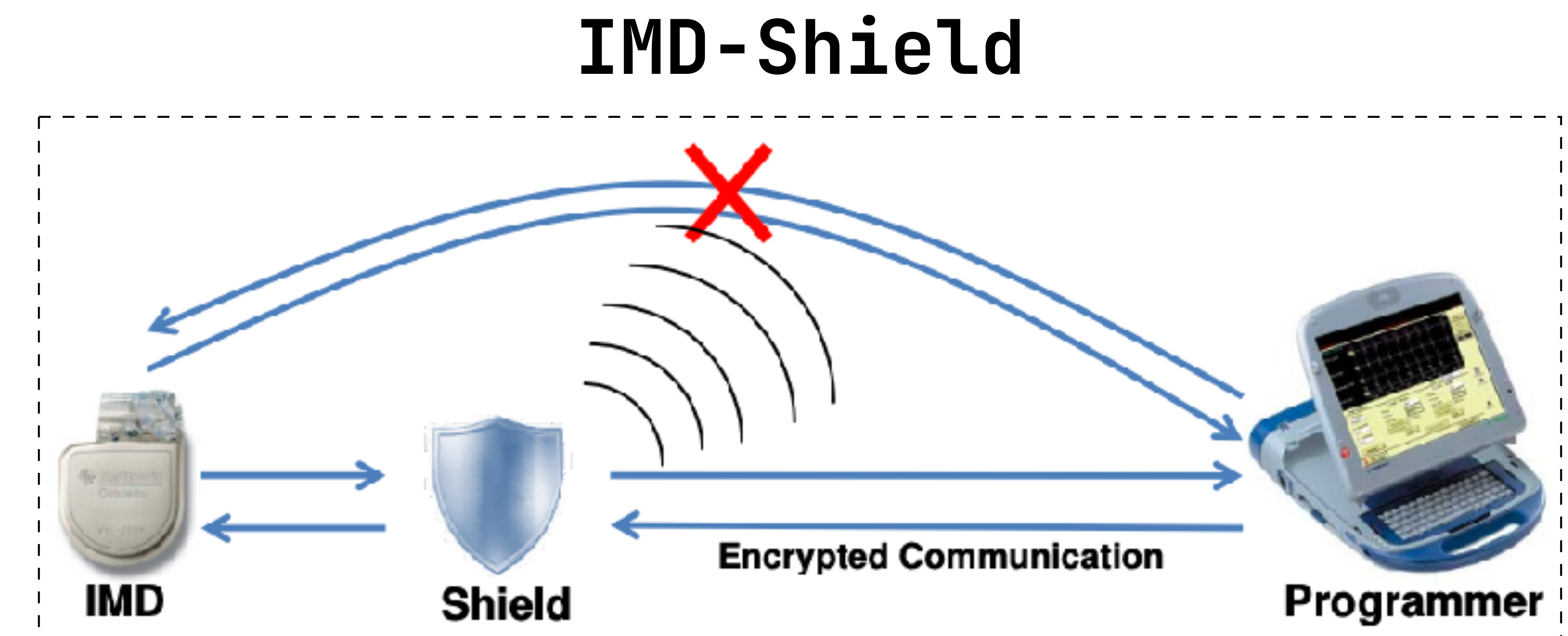
Authentication and Access Control



How It Works:

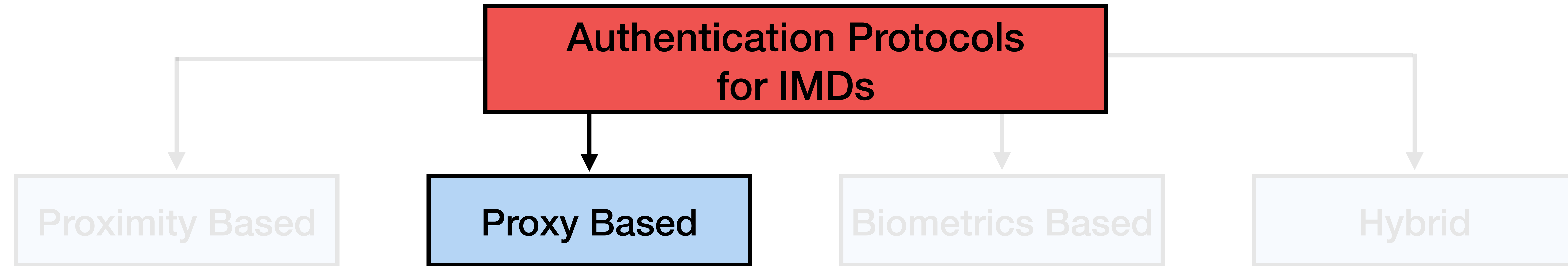
- A trusted **proxy** monitors IMD communication attempts
- It authenticates devices and blocks, filters or **mediates** unauthorized **access**

IMD-Shield



Gollakota, Shyamnath, et al. "They can hear your heartbeats: Non-invasive security for implantable medical devices." Proceedings of the ACM SIGCOMM 2011 conference. 2011.

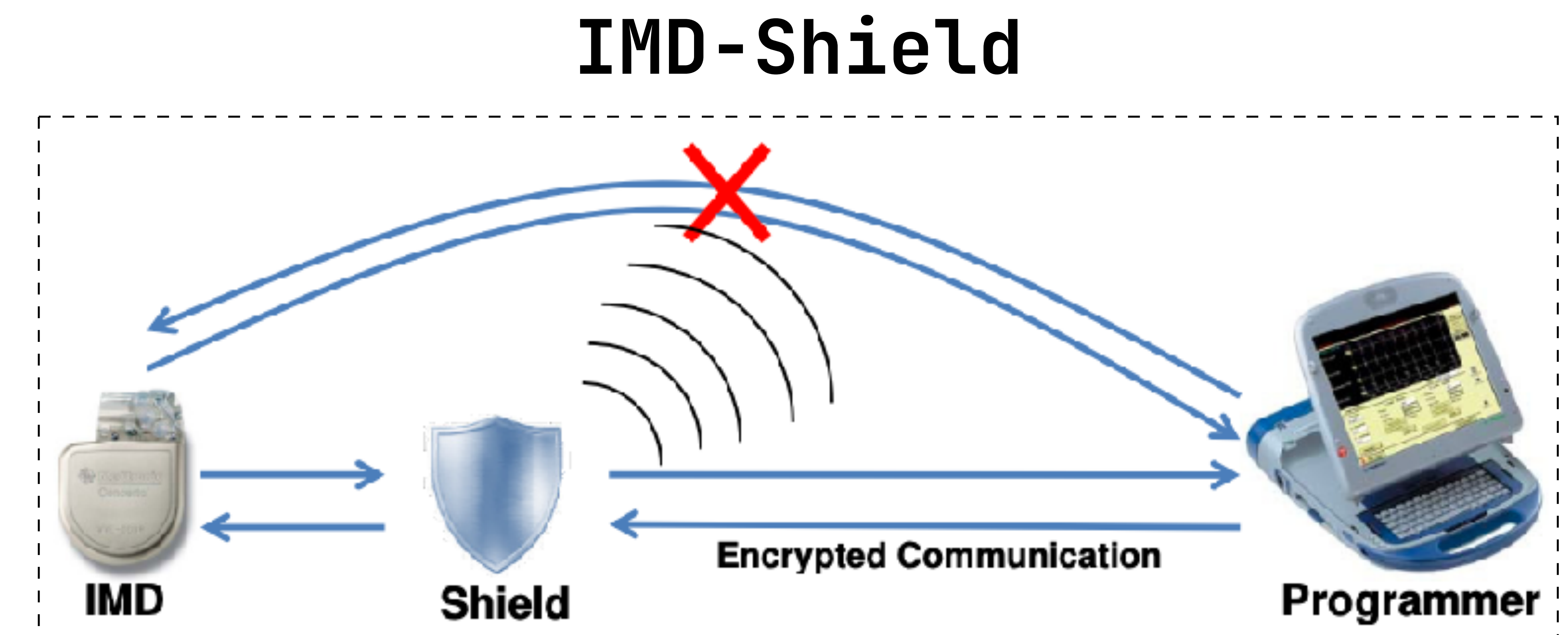
Authentication and Access Control



Assumptions:

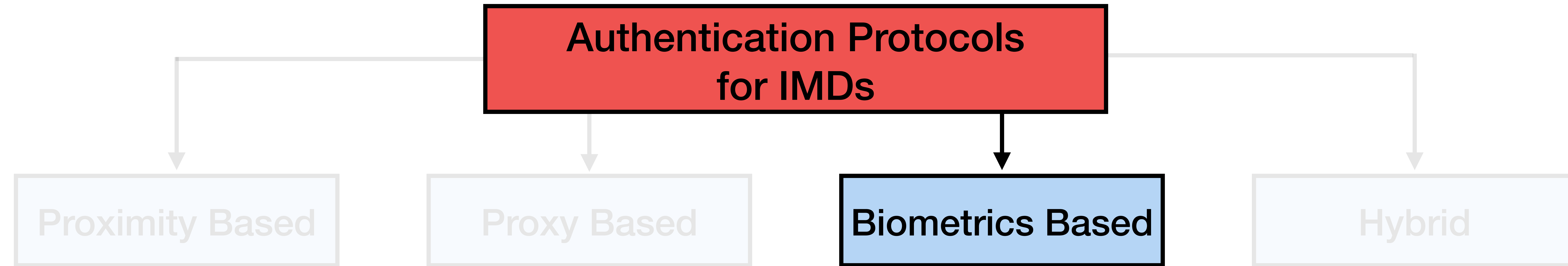
- The proxy is trusted and **remains close** to the patient and IMD
- Attackers cannot bypass, compromise, or disable the proxy

IMD-Shield



Gollakota, Shyamnath, et al. "They can hear your heartbeats: Non-invasive security for implantable medical devices." Proceedings of the ACM SIGCOMM 2011 conference. 2011.

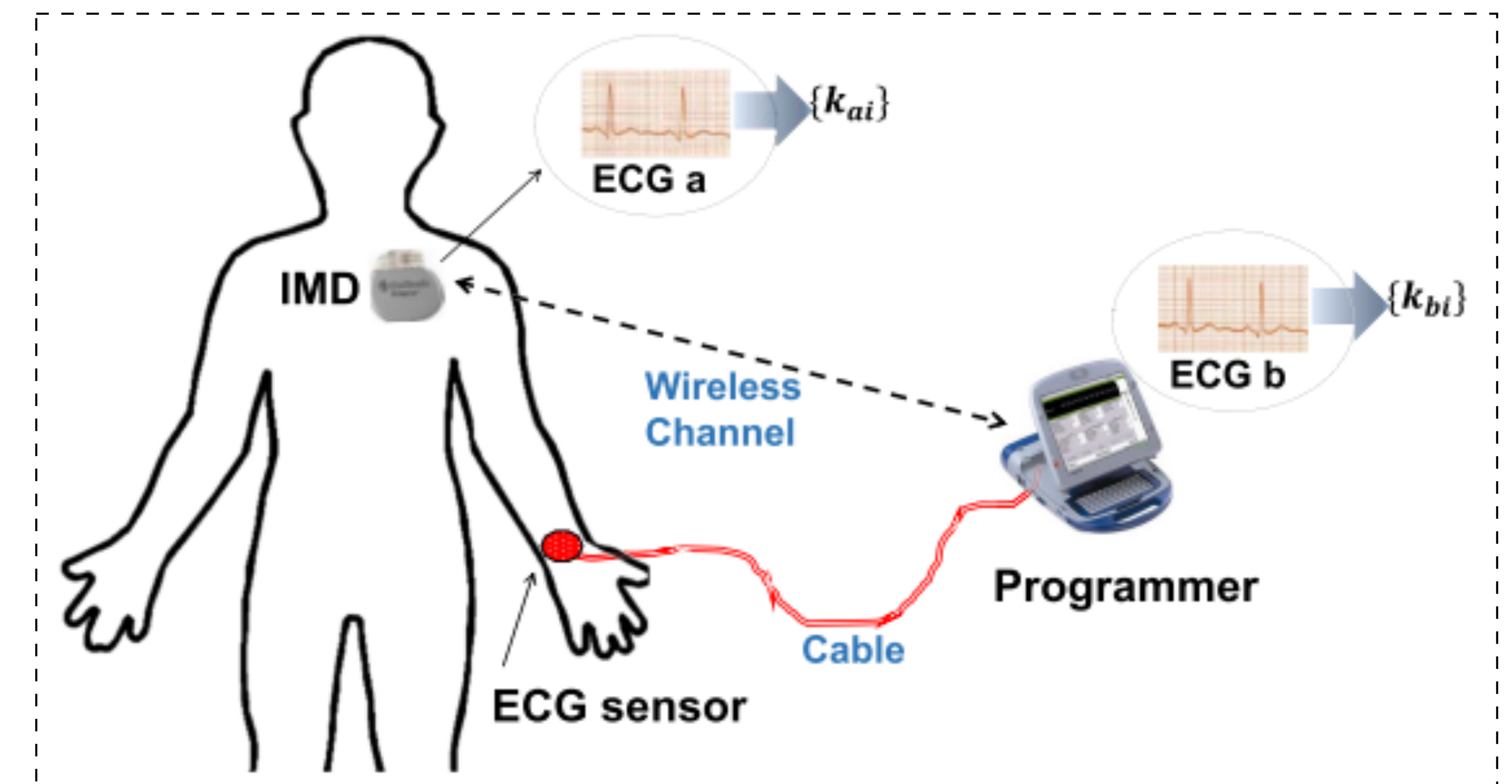
Authentication and Access Control



How They Work:

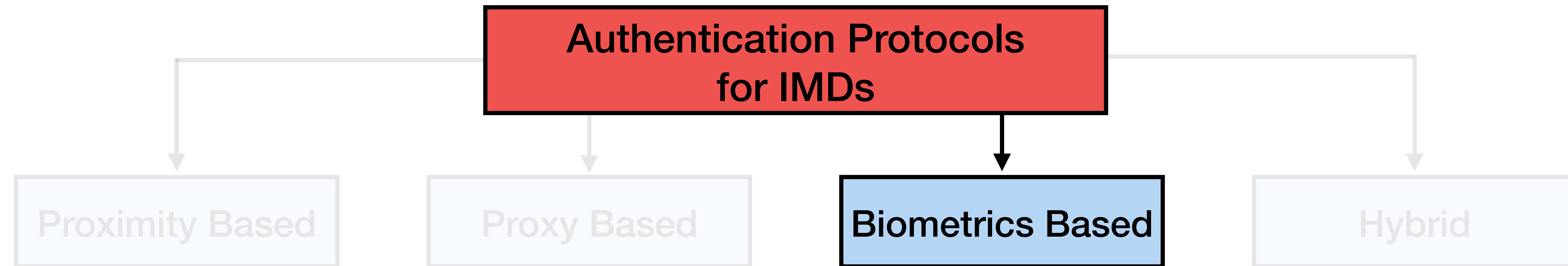
- Use patient-specific **physiological signals** as the authentication factor
- Both devices **derive** and compare **authentication material** from the measured signal

ECG-Based Authentication

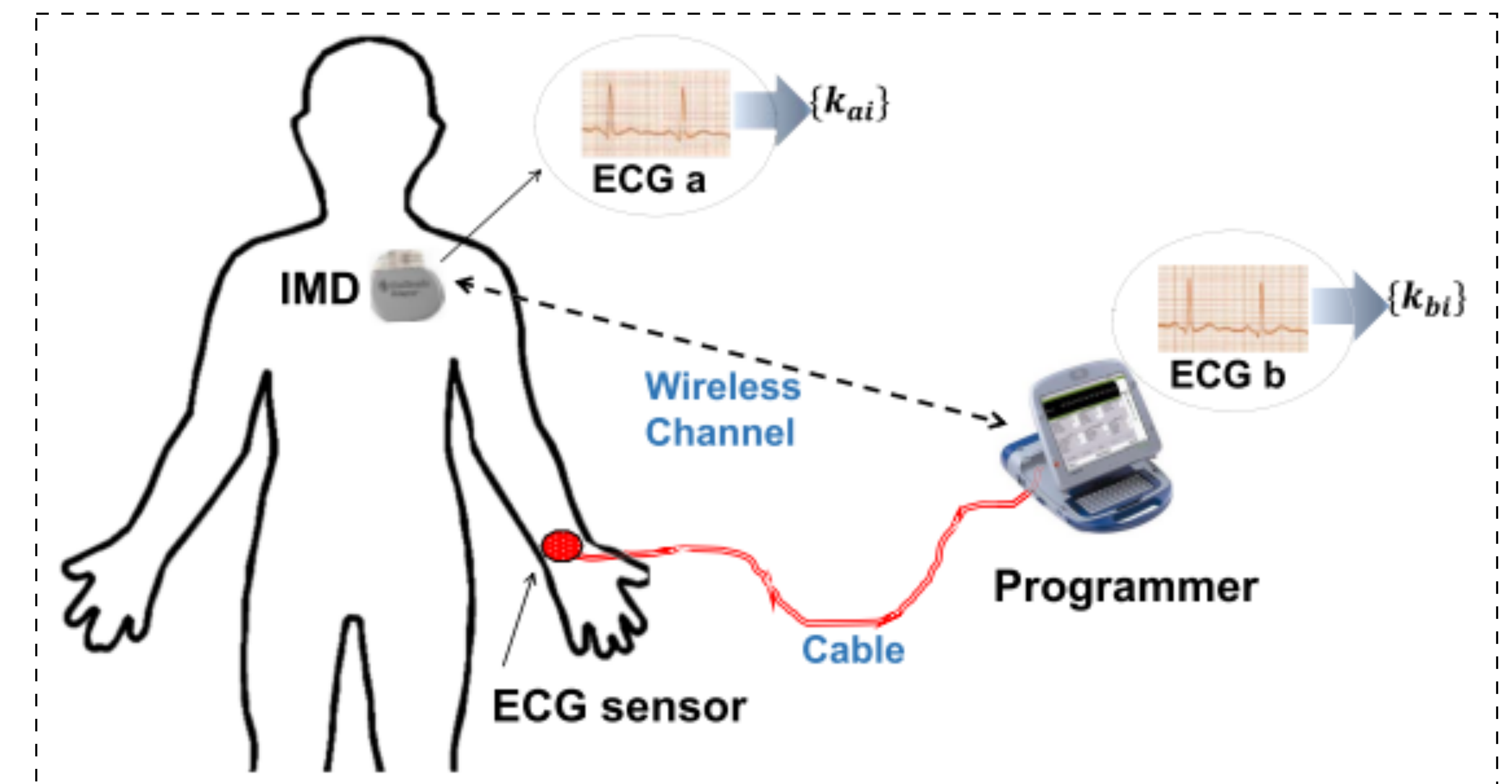


Rostami, Masoud, Ari Juels, and Farinaz Koushanfar. "Heart-to-heart (H2H) authentication for implanted medical devices." Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. 2013.

Authentication and Access Control



ECG-Based Authentication

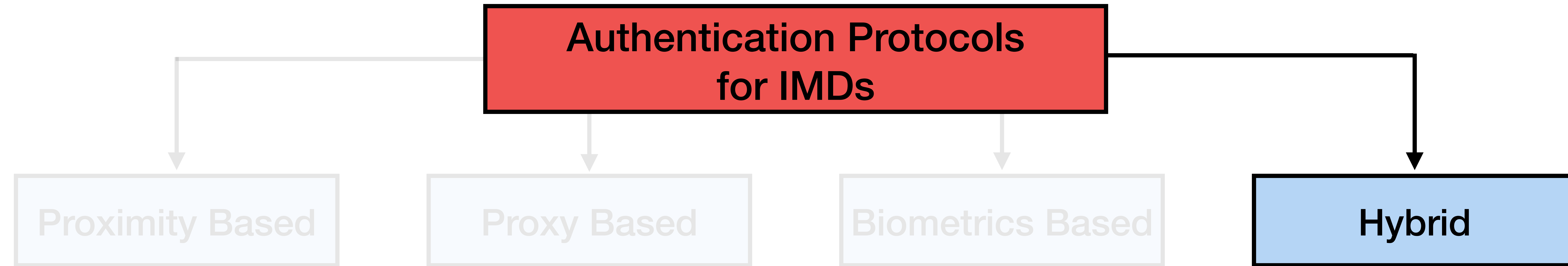


Assumptions:

- Physiological signals are **reliable, unique, and stable** enough for authentication
- **Attackers cannot** accurately **reproduce**, predict, or replay the signal

Rostami, Masoud, Ari Juels, and Farinaz Koushanfar. "Heart-to-heart (H2H) authentication for implanted medical devices." Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. 2013.

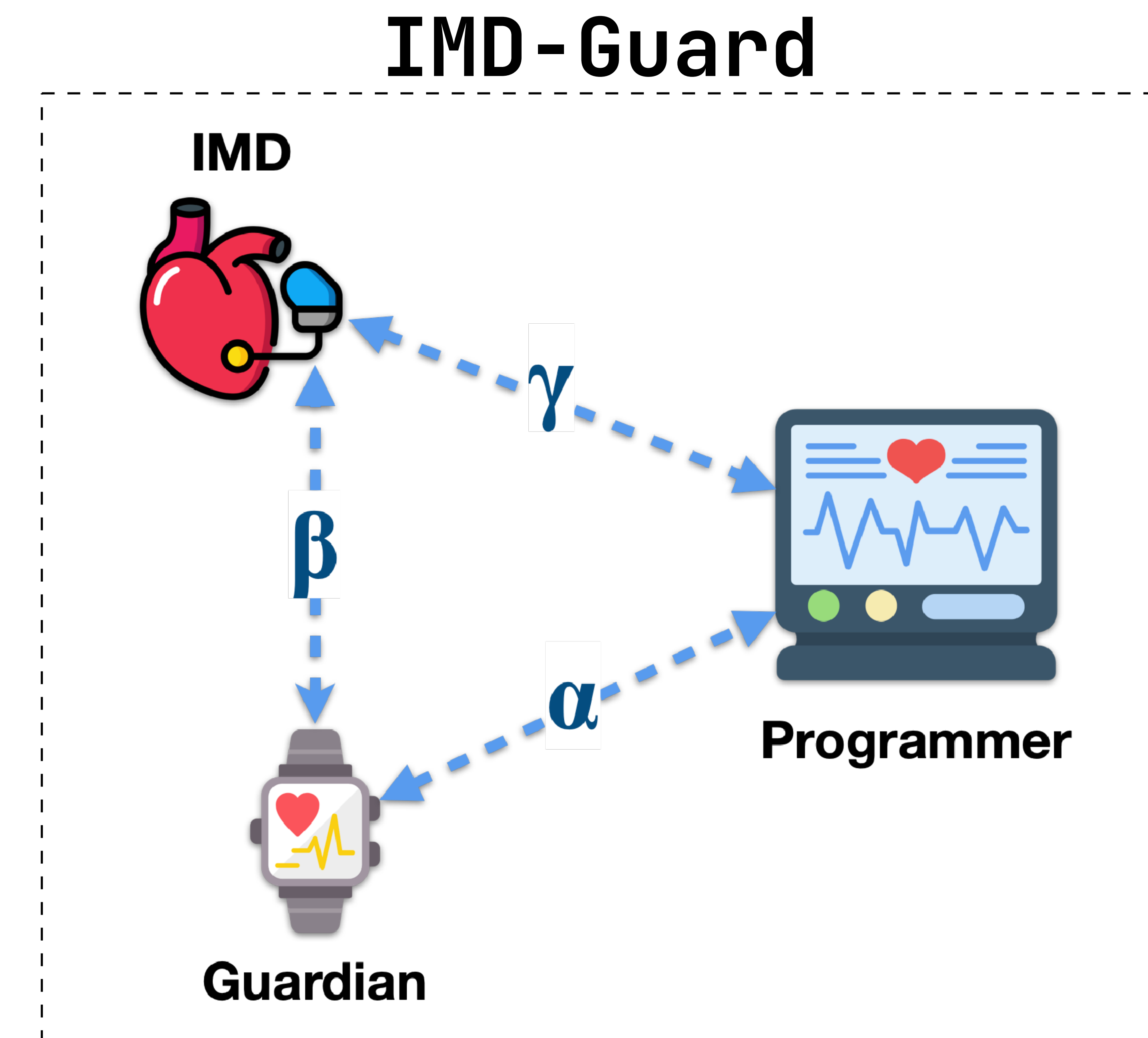
Authentication and Access Control



How They Work:

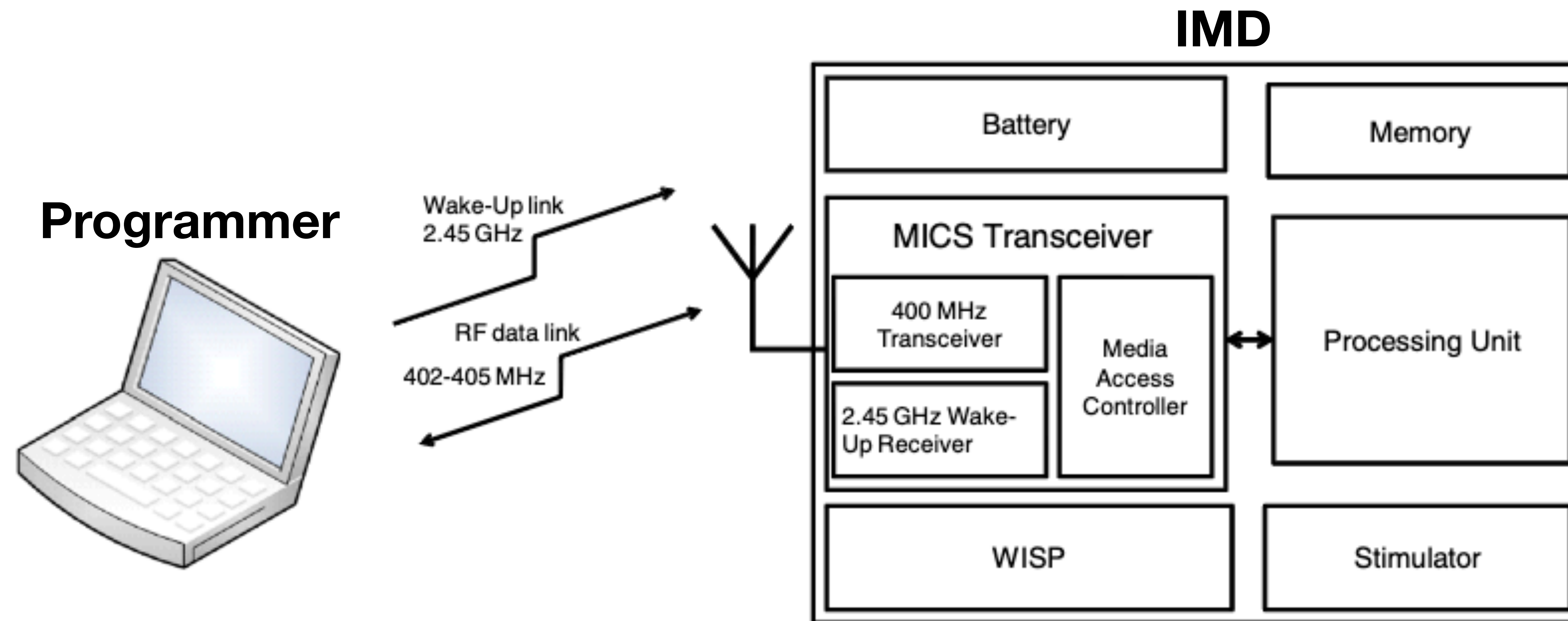
- Combine **multiple mechanisms**, such as proxy-based mediation and proximity verification

IMD-Guard



Xu, Fengyuan, et al. "IMDGuard: Securing implantable medical devices with the external wearable guardian." *2011 Proceedings IEEE INFOCOM*. IEEE, 2011

IMD Architecture & Constraints



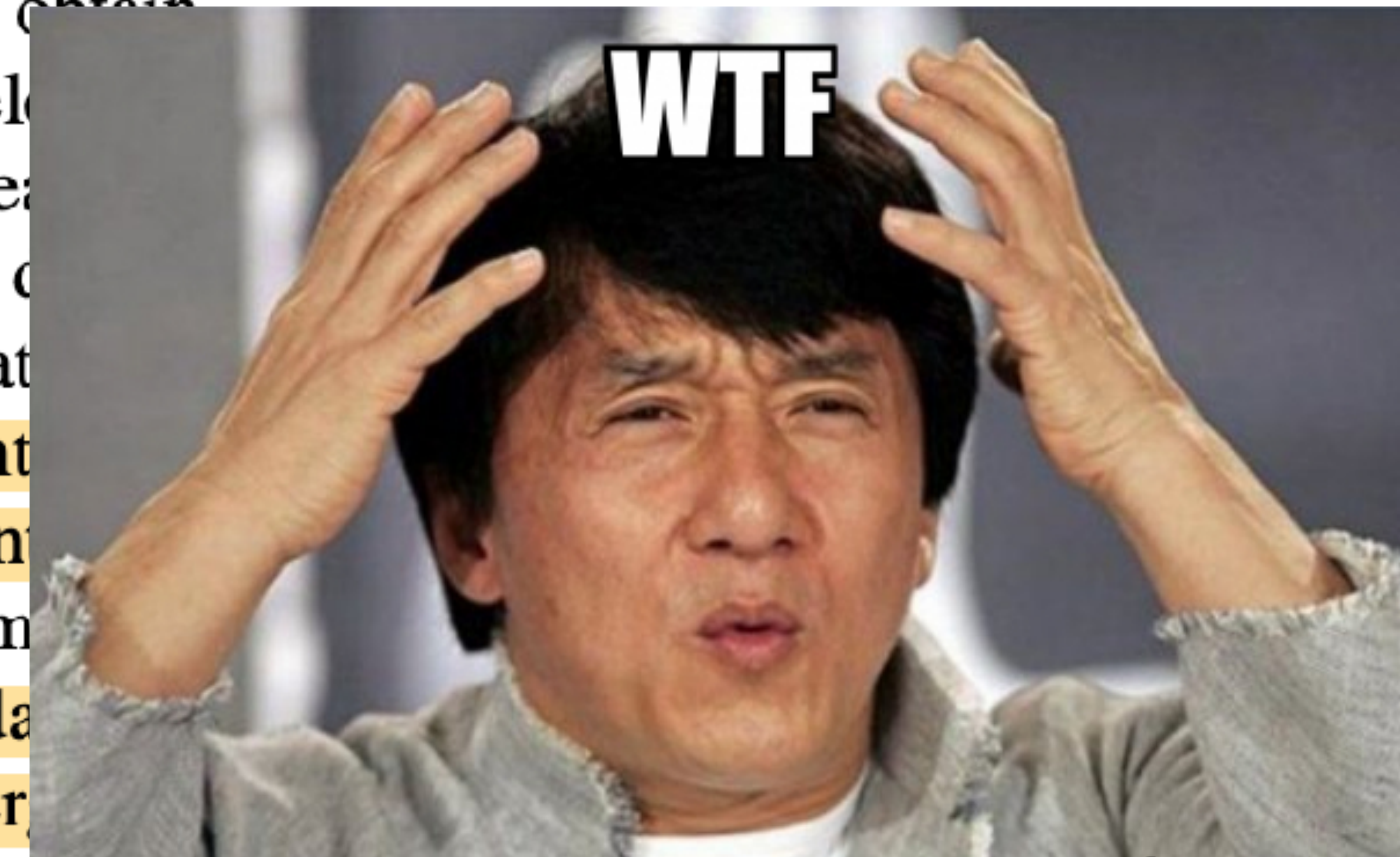
Trade-Offs:

- Security vs. Limited Resources
- Security vs. Accessibility
  - Authentication and Access-control
  - Emergency and Normal Use

Ellouze, Nourhene, et al. "Security of implantable medical devices: Limits, requirements, and proposals." Security and Communication Networks 7.12 (2014): 2475-2491.

Supporting Emergency Access

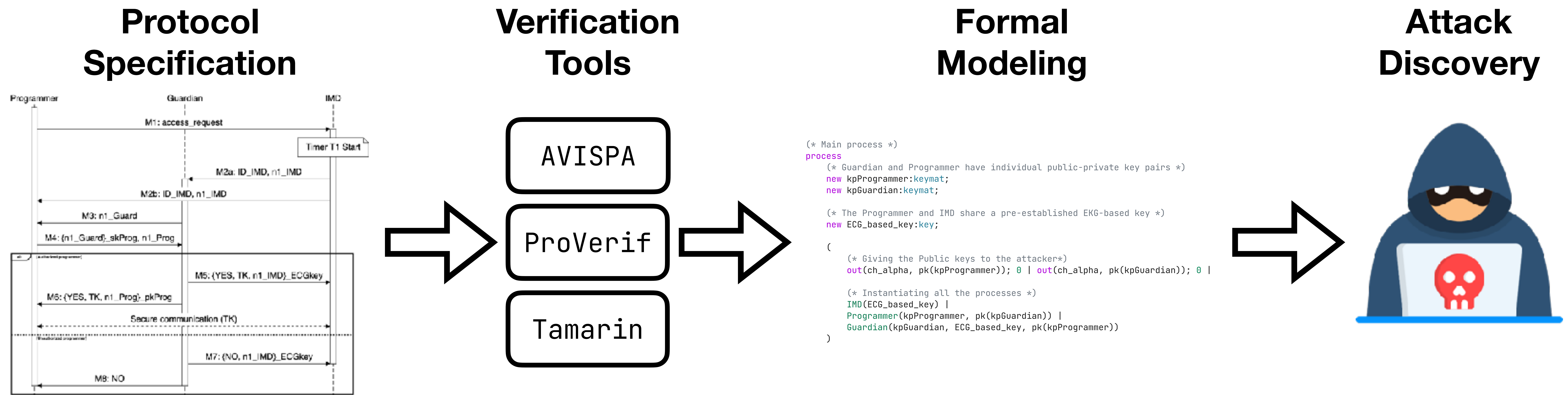
1) Direct-KD: A direct Key Distribution (Direct-KD)
method can be used to provide the key instantaneously during
the emergency situation by printing the key on a bracelet or the
patient's skin. Emergency doctors can read the key and
access to the IMD. The key may be engraved on a bracel
or stored in a smart card [46]. The patient has to wea
bracelet or bring the smart card and present it to the
in their normal visits or in case of an emergency treat
Alternatively, a visible or UV-visible tattoo can be print
the skin to represent a scannable password, so the patien
not need to wear anything [9], [10]. However, this m
is hard to revoke or reissue the key. Besides, some da
to the skin may hinder access to the IMD in the emer
situation. Another possible solution is to use a universal key
for the IMDs with the same model. But the adversary can also
discover the key through side-channel attacks or by hacking
into the doctor's computers [19].



Zheng, Guanglou, et al. "Ideas and challenges for securing wireless implantable medical devices: A review." IEEE Sensors Journal 17.3 (2016): 562-576.

Formal Verification

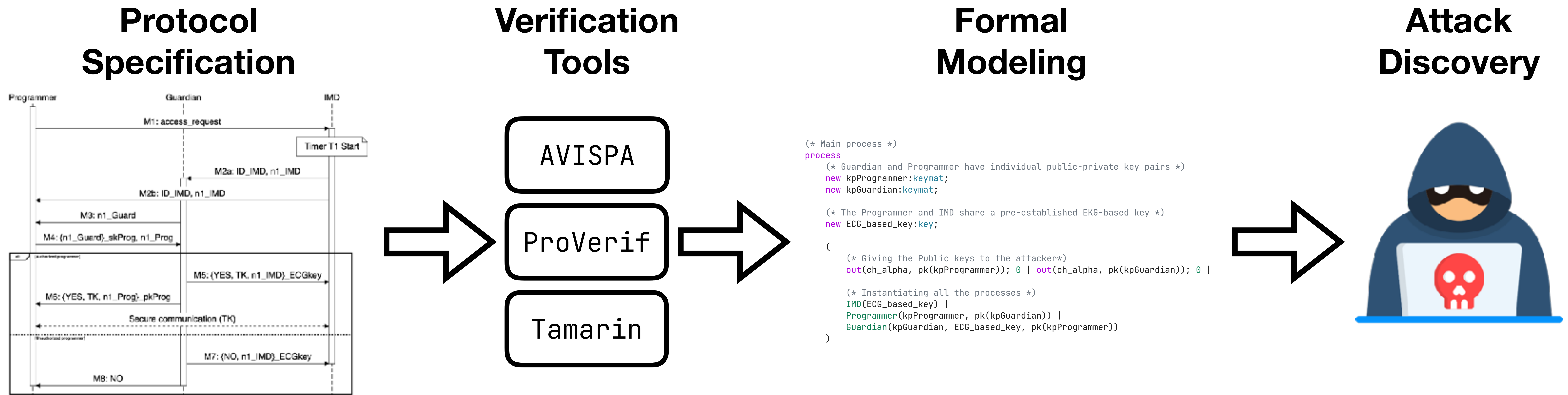
- **Mathematical approach** used to verify whether a system satisfies specific security or correctness properties.
- Systems are **formally modeled**, and analyzed against desired **security properties** using automated or mathematical verification techniques.



Formal Verification

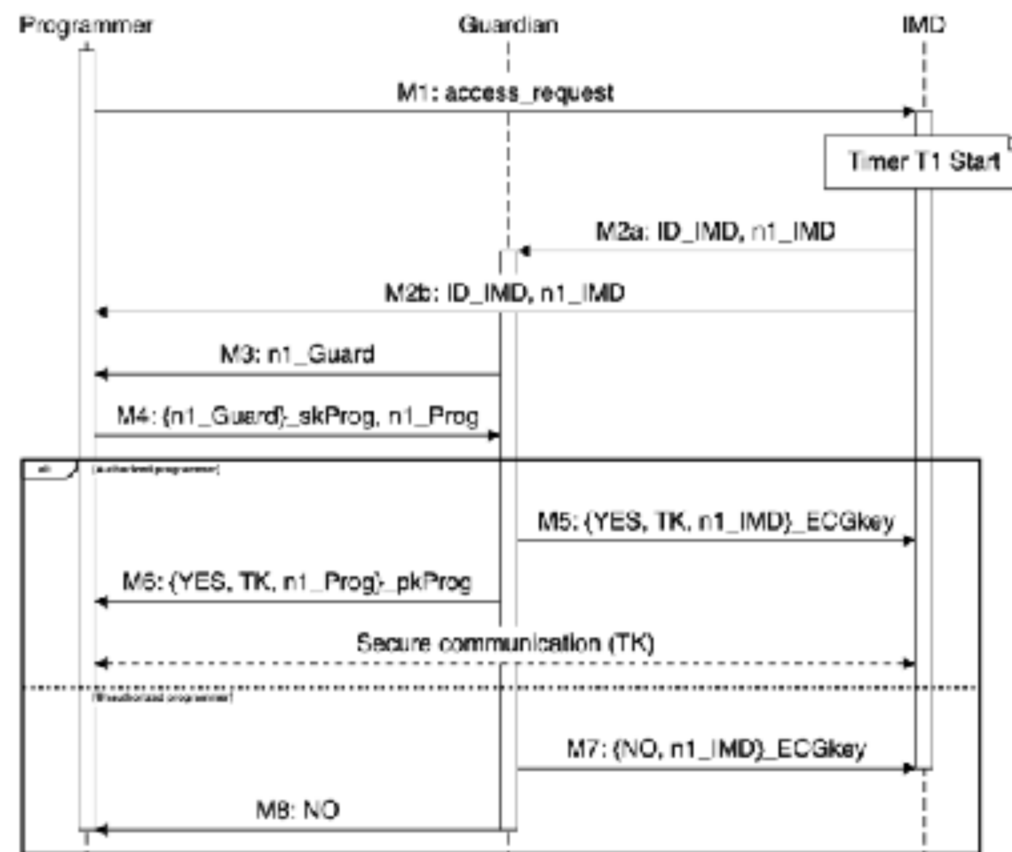
Why Use It?

- Security protocols are subtle and **error-prone**
- **Testing is limited**; formal methods provide rigorous analysis.



Protocol Specification

Protocol Specification



Verification Tools

Formal Modeling

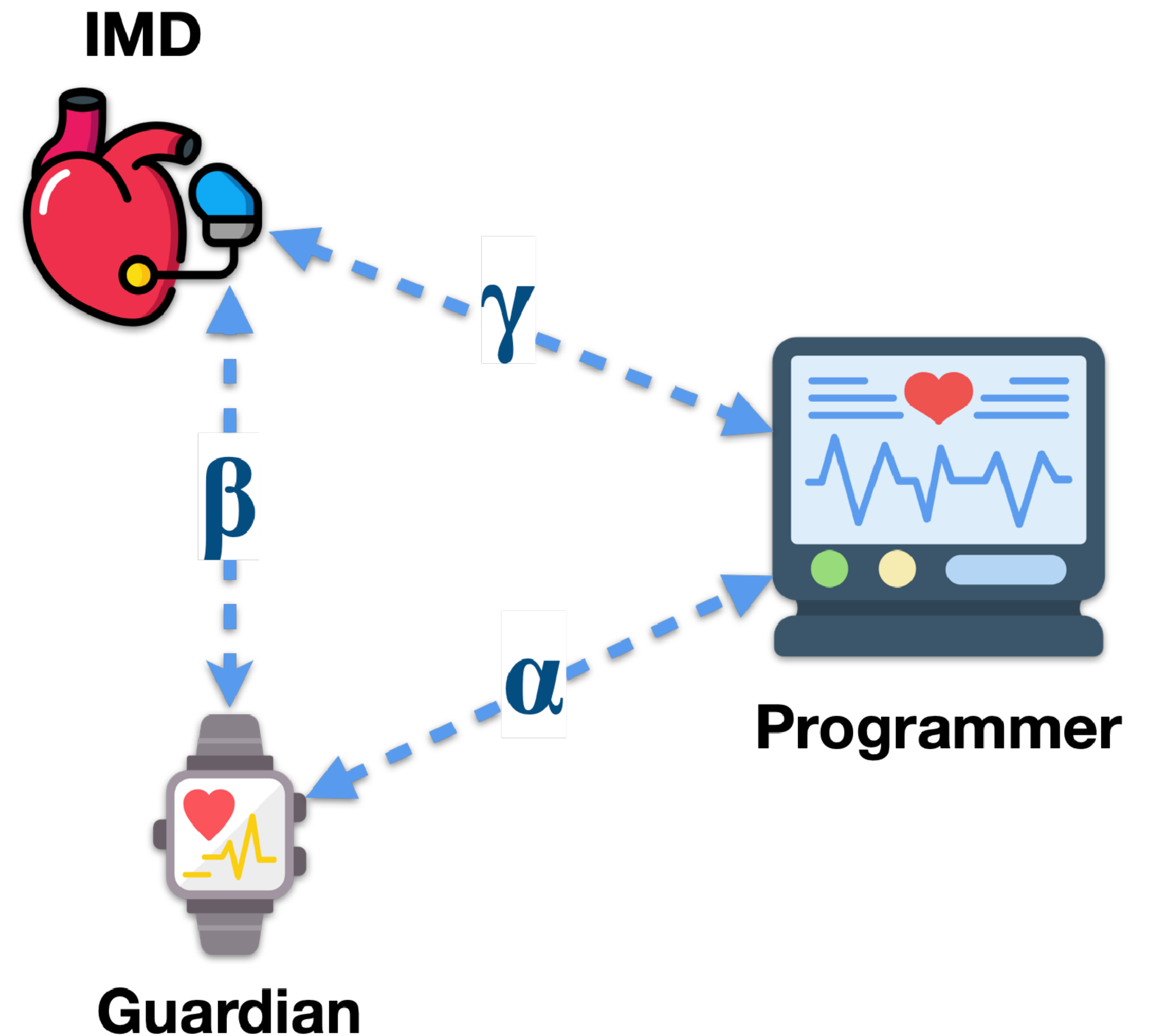
```
(* Main process *)
process
    (* Guardian and Programmer have individual public-private key pairs *)
    new kpProgrammer:keymat;
    new kpGuardian:keymat;

    (* The Programmer and IMD share a pre-established EKG-based key *)
    new ECG_based_key:key;

    (
        (* Giving the Public keys to the attacker*)
        out(ch_alpha, pk(kpProgrammer)); 0 | out(ch_alpha, pk(kpGuardian)); 0 |

        (* Instantiating all the processes *)
        IMD(ECG_based_key) |
        Programmer(kpProgrammer, pk(kpGuardian)) |
        Guardian(kpGuardian, ECG_based_key, pk(kpProgrammer))
    )
```

Attack Discovery

IMDGuard Protocol

Three Logical Links:

- **Link α**: for access control
  - Pre-installed Public Keys
- **Link β**: for initial pairing
  - ECG Key
- **Link γ**: to send commands to the IMD
  - Session Key



Xu, Fengyuan, et al. "IMDGuard: Securing implantable medical devices with the external wearable guardian." *2011 Proceedings IEEE INFOCOM*. IEEE, 2011

IMDGuard Protocol

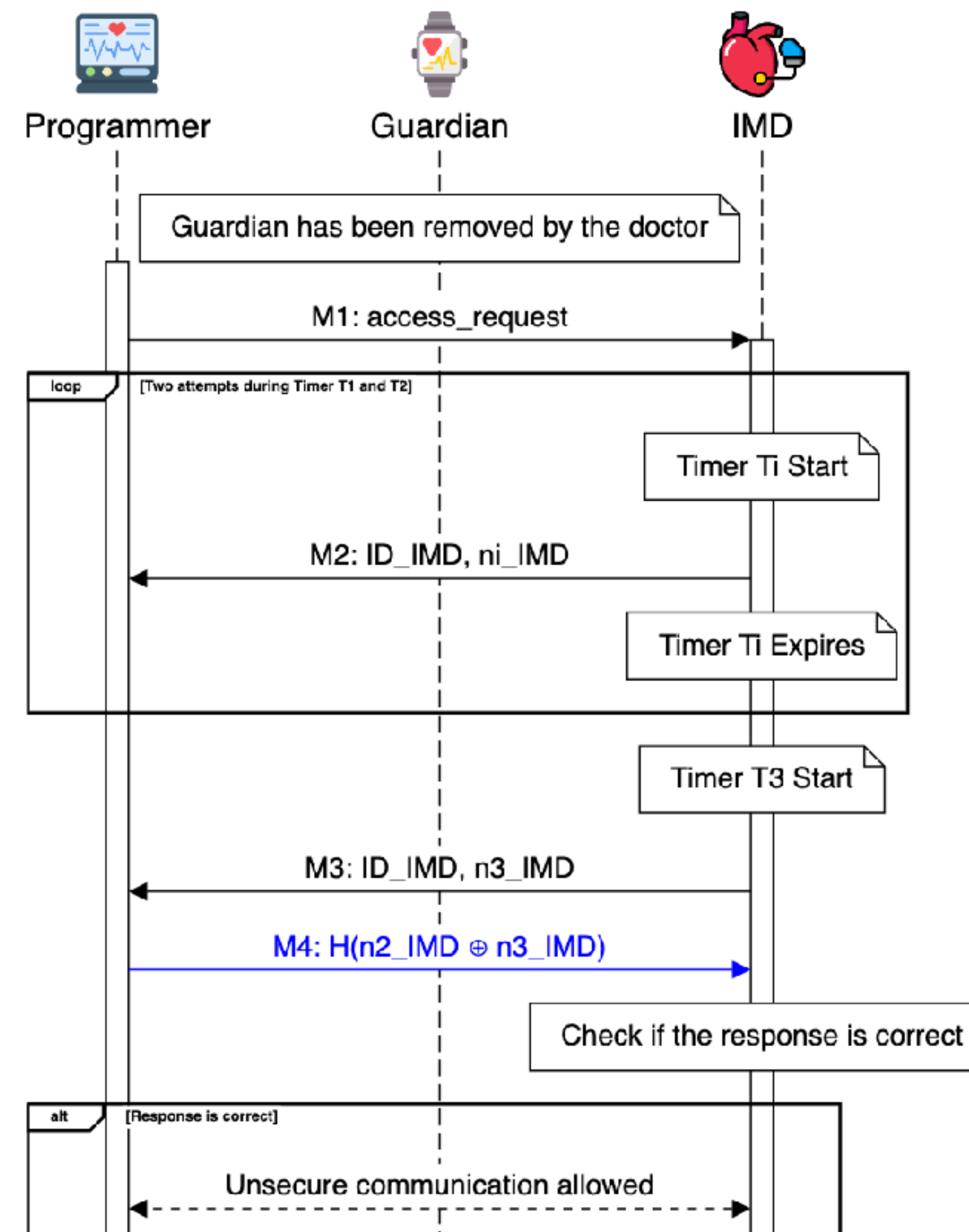
*If the IMD receives the authentication result before the T1 timer expires, the system operates in **Regular Mode**.*

Programmer
Guardian
IMD
M1: access_request
Timer T1 Start
M2a: ID_IMD, n1_IMD
M2b: ID_IMD, n1_IMD
M3: n1_Guard
M4: {n1_Guard}_skProg, n1_Prog
alt
[Authorized programmer]
M5: {YES, TK, n1_IMD}_ECGkey
M6: {YES, TK, n1_Prog}_pkProg
Secure communication (TK)
[Unauthorized programmer]
M7: {NO, n1_IMD}_ECGkey
M8: NO

Xu, Fengyuan, et al. "IMDGuard: Securing implantable medical devices with the external wearable guardian." *2011 Proceedings IEEE INFOCOM*. IEEE, 2011

IMDGuard Protocol

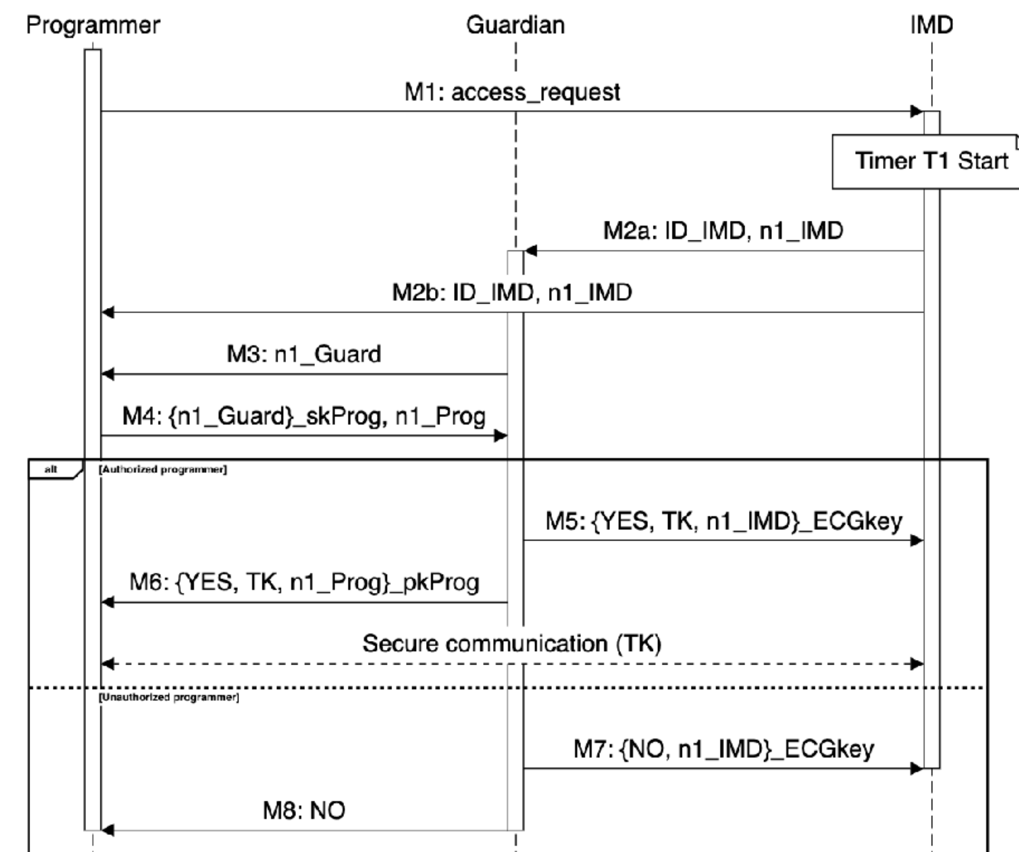
If the authentication procedure repeatedly fails and the timer expires three times, the protocol automatically ***downgrades*** *to* ***Emergency Mode***.



Xu, Fengyuan, et al. "IMDGuard: Securing implantable medical devices with the external wearable guardian." *2011 Proceedings IEEE INFOCOM*. IEEE, 2011

Verification Tools

Protocol Specifications



Verification Tools

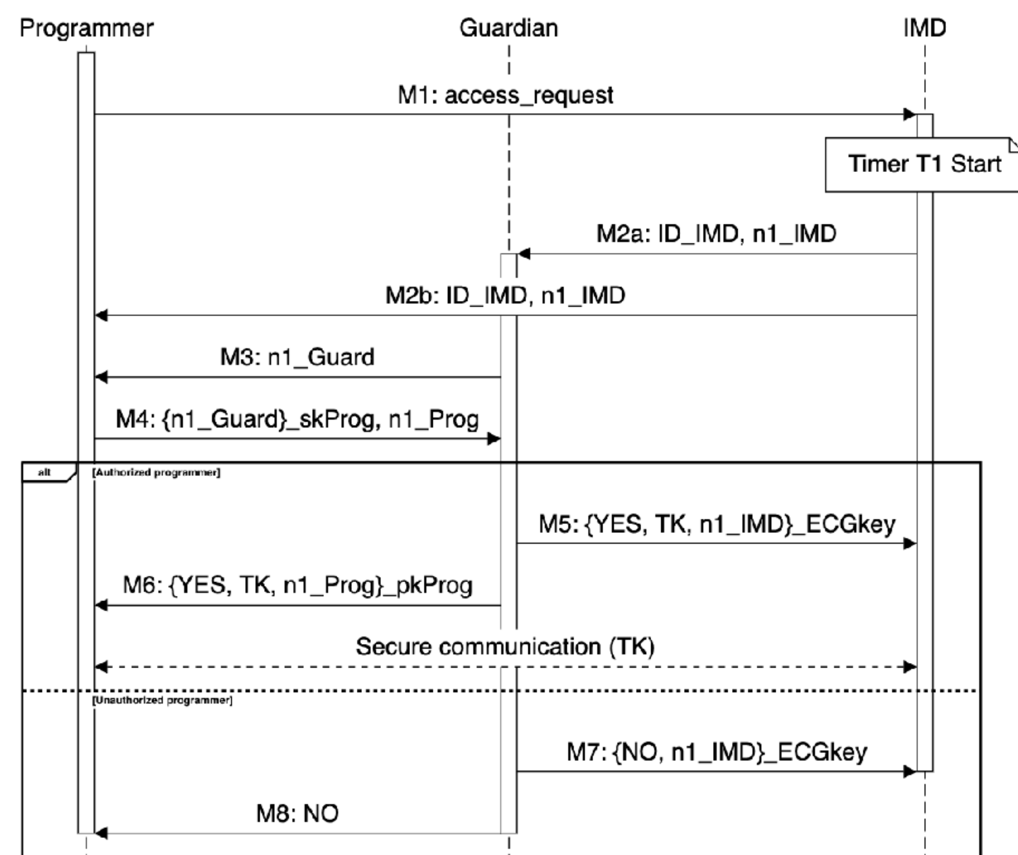
Formal Modeling

```
(* Main process *)
process
    (* Guardian and Programmer have individual public-private key pairs *)
    new kpProgrammer:keymat;
    new kpGuardian:keymat;

    (* The Programmer and IMD share a pre-established EKG-based key *)
    new ECG_based_key:key;

    (
        (* Giving the Public keys to the attacker*)
        out(ch_alpha, pk(kpProgrammer)); 0 | out(ch_alpha, pk(kpGuardian)); 0 |

        (* Instantiating all the processes *)
        IMD(ECG_based_key) |
        Programmer(kpProgrammer, pk(kpGuardian)) |
        Guardian(kpGuardian, ECG_based_key, pk(kpProgrammer))
    )
```

Attack Discovery

Verification Tools

Protocol Specifications



Verification Tools

Formal Modeling

```
(* Main process *)
process
    (* Guardian and Programmer have individual public-private key pairs *)
    new kpProgrammer:keymat;
    new kpGuardian:keymat;

    (* The Programmer and IMD share a pre-established EKG-based key *)
    new ECG_based_key:key;

    (
        (* Giving the Public keys to the attacker*)
        out(ch_alpha, pk(kpProgrammer)); 0 | out(ch_alpha, pk(kpGuardian)); 0 |

        (* Instantiating all the processes *)
        IMD(ECG_based_key) |
        Programmer(kpProgrammer, pk(kpGuardian)) |
        Guardian(kpGuardian, ECG_based_key, pk(kpProgrammer))
    )
```

Attack Discovery

ProVerif

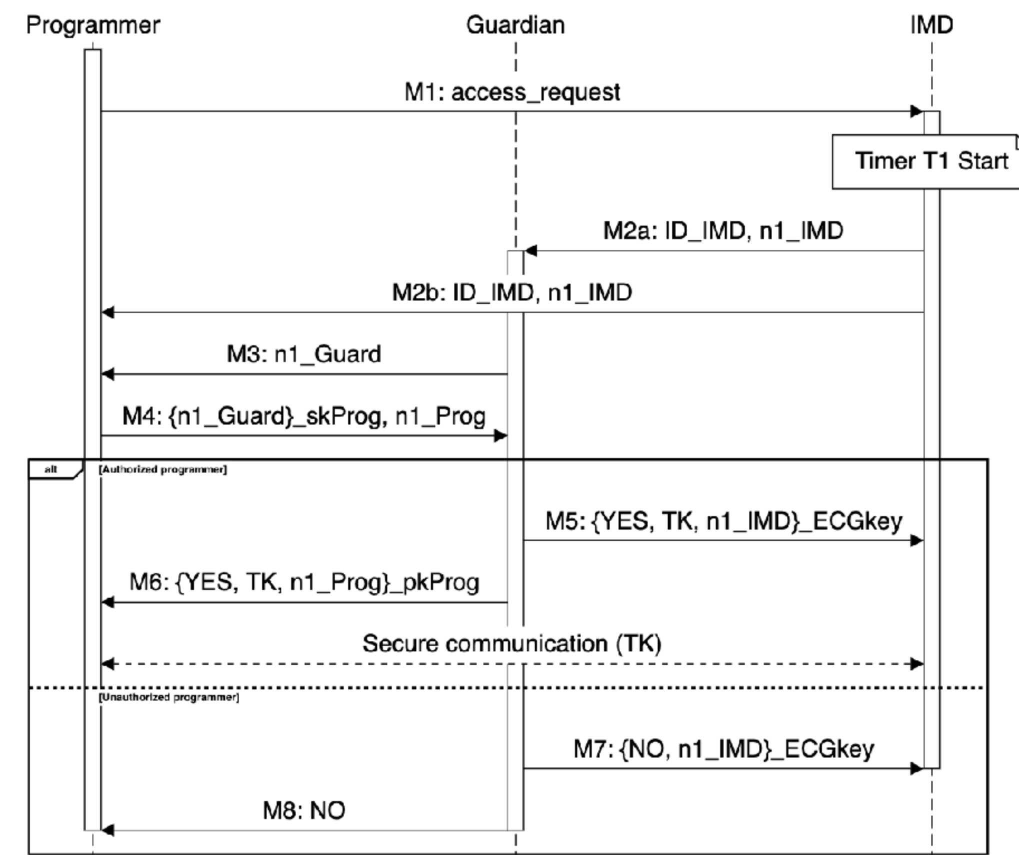
- Formal verification tool for security protocols
- Detects **flaws** and generates **attack traces** when properties fail
- Based on symbolic **model checking** and the **applied pi-calculus**
- Assumes **perfect cryptography**
- **Dolev-Yao attacker**: can eavesdrop, modify, replay, forge, and inject messages; can decrypt only with known keys.

```
process
  (* G and P each have a private-public key pair *)
  new kpProg:keymat;
  new kpGuard:keymat;

  (* Assume P and the IMD already share a key *)
  new ECG_based_key:key;

  (
    (* Give the public keys to the attacker *)
    out(ch_alpha, pk(kpProg)); 0  |
    out(ch_alpha, pk(kpGuard)); 0 |

    (* Instantiate all processes *)
    IMD(ECG_based_key)  |
    Programmer(kpProg, pk(kpGuard))  |
    Guardian(kpGuard, ECG_based_key, pk(kpProg))
  )
```

Blanchet, Bruno. "Modeling and verifying security protocols with the applied pi calculus and ProVerif." Foundations and Trends in Signal Processing 1.1-2 (2016): 1-135.

Formal Modeling

Protocol Specifications



Verification Tools

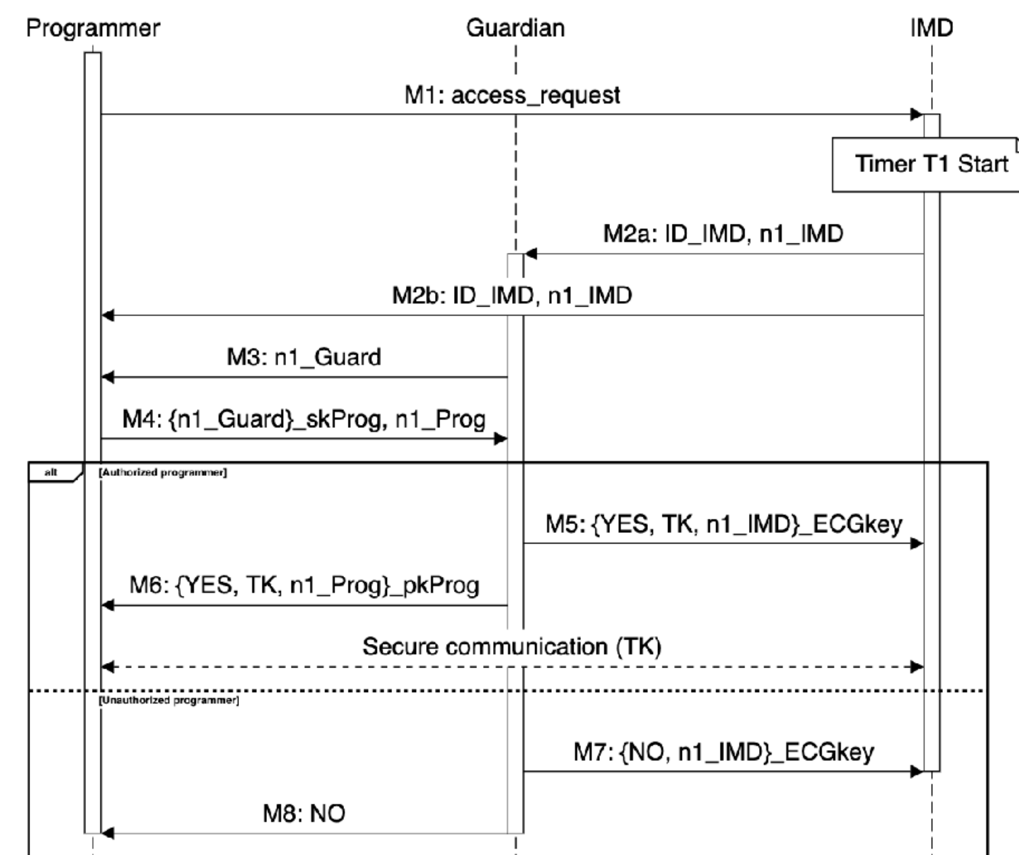
Formal Modeling

```
(* Main process *)
process
    (* Guardian and Programmer have individual public-private key pairs *)
    new kpProgrammer:keymat;
    new kpGuardian:keymat;

    (* The Programmer and IMD share a pre-established EKG-based key *)
    new ECG_based_key:key;

    (
        (* Giving the Public keys to the attacker*)
        out(ch_alpha, pk(kpProgrammer)); 0 | out(ch_alpha, pk(kpGuardian)); 0 |

        (* Instantiating all the processes *)
        IMD(ECG_based_key) |
        Programmer(kpProgrammer, pk(kpGuardian)) |
        Guardian(kpGuardian, ECG_based_key, pk(kpProgrammer))
    )
```

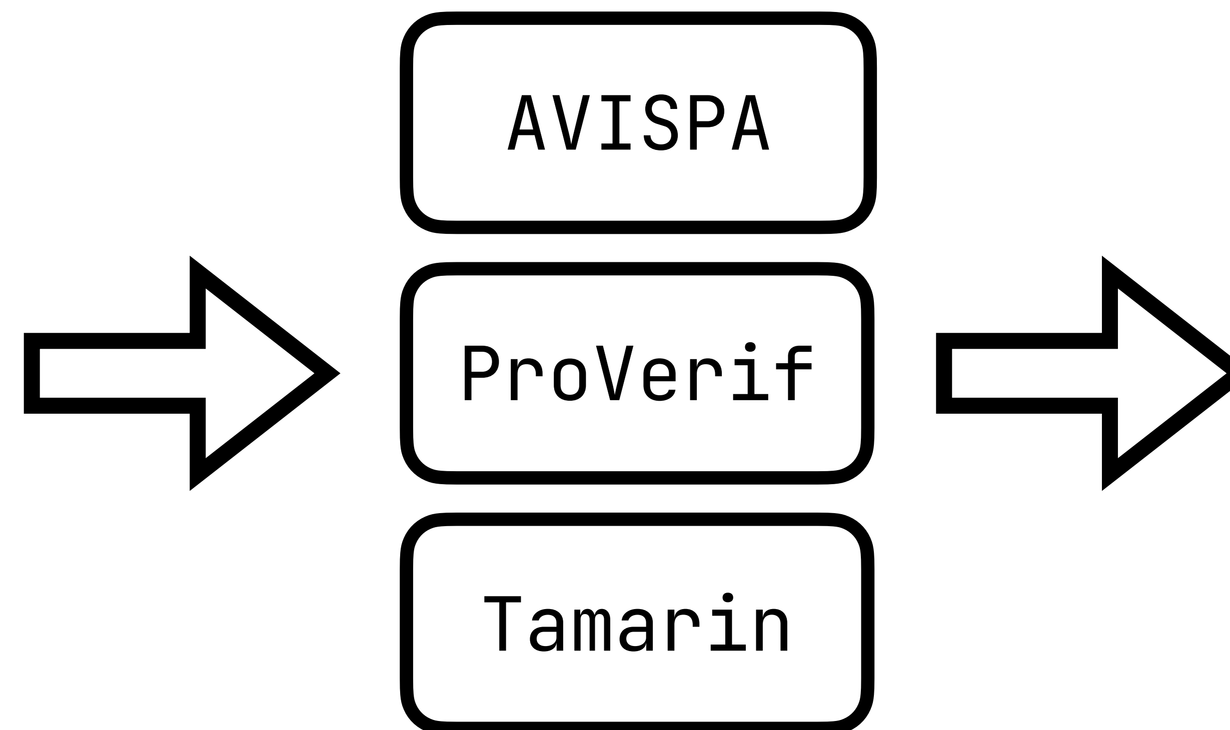
Attack Discovery

Formal Modeling

Protocol Specifications



Verification Tools



Formal Modeling

```
(* Main process *)
process
    (* Guardian and Programmer have individual public-private key pairs *)
    new kpProgrammer:keymat;
    new kpGuardian:keymat;

    (* The Programmer and IMD share a pre-established EKG-based key *)
    new ECG_based_key:key;

    (
        (* Giving the Public keys to the attacker*)
        out(ch_alpha, pk(kpProgrammer)); 0 | out(ch_alpha, pk(kpGuardian)); 0 |

        (* Instantiating all the processes *)
        IMD(ECG_based_key) |
        Programmer(kpProgrammer, pk(kpGuardian)) |
        Guardian(kpGuardian, ECG_based_key, pk(kpProgrammer))
    )
```

Attack Discovery

Threat Model

Known to the Adversary:

- Public keys of *G* and *P*
- Message formats (*Initialization*, *YES*, *NO*)
- Nonces and terms after cleartext or wrong encryption exposure

NOT Known to the Adversary:

- Private keys
- Shared secret keys
- Nonces and terms before exposure

Programmer
Guardian
IMD
M1: access_request
Timer T1 Start
M2a: ID_IMD, n1_IMD
M2b: ID_IMD, n1_IMD
M3: n1_Guard
M4: {n1_Guard}_skProg, n1_Prog
alt
[Authorized programmer]
M5: {YES, TK, n1_IMD}_ECGkey
M6: {YES, TK, n1_Prog}_pkProg
Secure communication (TK)
[Unauthorized programmer]
M7: {NO, n1_IMD}_ECGkey
M8: NO

Xu, Fengyuan, et al. "IMDGuard: Securing implantable medical devices with the external wearable guardian." *2011 Proceedings IEEE INFOCOM*. IEEE, 2011

Verification Goals

- **Confidentiality:** Prevent eavesdropping from revealing sensitive medical data.
- **Authenticity & Integrity:** Prevent impersonation and message manipulation.
- **Availability:** Detect protocol-level denial-of-service conditions.

Programmer
Guardian
IMD
M1: access_request
Timer T1 Start
M2a: ID_IMD, n1_IMD
M2b: ID_IMD, n1_IMD
M3: n1_Guard
M4: {n1_Guard}_skProg, n1_Prog
alt
[Authorized programmer]
M5: {YES, TK, n1_IMD}_ECGkey
M6: {YES, TK, n1_Prog}_pkProg
Secure communication (TK)
[Unauthorized programmer]
M7: {NO, n1_IMD}_ECGkey
M8: NO

Xu, Fengyuan, et al. "IMDGuard: Securing implantable medical devices with the external wearable guardian." *2011 Proceedings IEEE INFOCOM*. IEEE, 2011

Modeling IMDGuard in ProVerif

```
(* New datatypes *)
type key.      (* Symmetric key *)
type pkey.     (* Public key *)
type skey.     (* Private key *)
type keymat.   (* Key material *)
type result.   (* Result of signature check *)
```

> Built-in types in ProVerif: bitstring, channel, bool

```
(* Publicly known constants and channels (all are accessible by the attacker) *)
free access_request:bitstring.
free YES:bitstring.
free NO:bitstring.
```

> free: Declares a publicly known 'name'

```
free ch_alpha:channel.    (* between Guardian and Programmer *)
free ch_beta:channel.     (* between Guardian and IMD *)
free ch_gamma:channel.    (* between IMD and Programmer *)
```

> [private]: Declares a secret name

```
(* Values NOT known by the attacker *)
free command:bitstring [private].          (* Command sent by the Programmer *)
free sensitive_data:bitstring [private].   (* PHI sent by the IMD *)
```

Modeling IMDGuard in ProVerif

In ProVerif, we don't implement crypto. We model its ideal behavior using functions and equations.

```
(* Public-key Encryption *)
fun penc(bitstring, pkey): bitstring.
fun pk(keymat): pkey.
fun sk(keymat): skey.
reduc forall x:bitstring, y:keymat; pdec(penc(x,pk(y)),sk(y)) = x.

(* Shared-key cryptography *)
fun senc(bitstring, key): bitstring.
reduc forall x: bitstring, y: key; sdec(senc(x,y),y) = x.

(* Signatures *)
fun ok():result.
fun sign(bitstring, skey): bitstring.
reduc forall m:bitstring, y:keymat; checksign(sign(m,sk(y)), pk(y)) = ok().
reduc forall m:bitstring, y:keymat; getmess(sign(m,sk(y))) = m.
```

Modeling IMDGuard in ProVerif

> The *reduc* rules define the only way cryptographic operations can be reversed.

```
(* Public-key Encryption *)
fun penc(bitstring, pkey): bitstring.
fun pk(keymat): pkey.
fun sk(keymat): skey.
reduc forall x:bitstring, y:keymat; pdec(penc(x,pk(y)),sk(y)) = x.

(* Shared-key cryptography *)
fun senc(bitstring, key): bitstring.
reduc forall x: bitstring, y: key; sdec(senc(x,y),y) = x.

(* Signatures *)
fun ok():result.
fun sign(bitstring, skey): bitstring.
reduc forall m:bitstring, y:keymat; checksign(sign(m,sk(y)), pk(y)) = ok().
reduc forall m:bitstring, y:keymat; getmess(sign(m,sk(y))) = m.
```

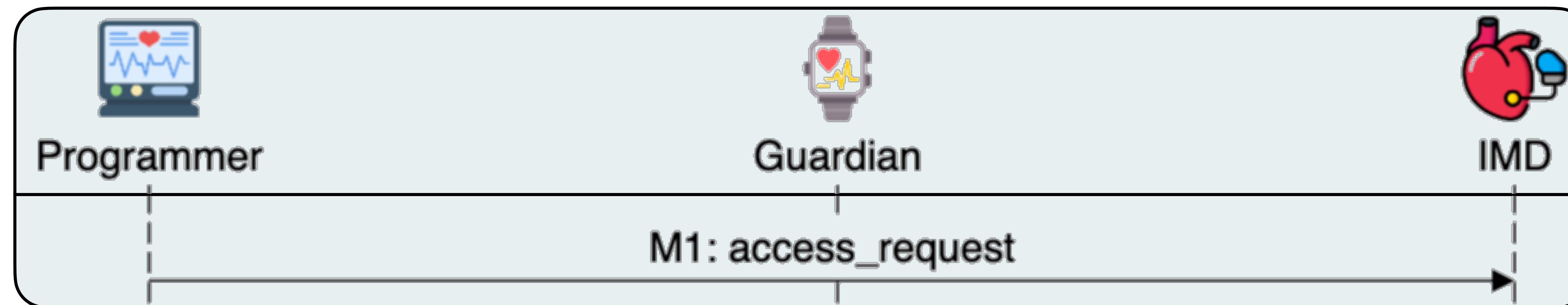
Modeling IMDGuard in ProVerif

```
(* Main process *)
process
    (* Guardian and Programmer have individual public-private key pairs *)
    new kpProgrammer:keymat;
    new kpGuardian:keymat;

    (* The Programmer and IMD share a pre-established EKG-based key *)
    new ECG_based_key:key;

    (
        (* Giving the Public keys to the attacker*)
        out(ch_alpha, pk(kpProgrammer)); 0 | out(ch_alpha, pk(kpGuardian)); 0 |

        (* Instantiating all the processes *)
        IMD(ECG_based_key) |
        Programmer(kpProgrammer, pk(kpGuardian)) |
        Guardian(kpGuardian, ECG_based_key, pk(kpProgrammer))
    )
```

Modeling the Guardian

```
let Guardian(kpGuardian:keymat, ECG_based_key:key, pkProgrammer:pkey) =
    in(ch_beta, (xID_IMD:bitstring, xn1_IMD:bitstring));

    new n1_Guard:bitstring;
    out(ch_alpha, n1_Guard);
    in(ch_alpha, (signature:bitstring, xn1_Prog:bitstring));

    if checksign(signature, pkProgrammer)=ok() then
    if n1_Guard = getmess(signature) then
        new temp_key:key;
        out(ch_beta, senc((YES, temp_key, xn1_IMD), ECG_based_key))
        out(ch_alpha, penc((YES, temp_key, xn1_Prog), pkProgrammer));

    0.
```

Modeling the Guardian




```
let Guardian(kpGuardian:keymat, ECG_based_key:key, pkProgrammer:pkey) =
    in(ch_beta, (xID_IMD:bitstring, xn1_IMD:bitstring));

    new n1_Guard:bitstring;
    out(ch_alpha, n1_Guard);
    in(ch_alpha, (signature:bitstring, xn1_Prog:bitstring));

    if checksign(signature, pkProgrammer)=ok() then
    if n1_Guard = getmess(signature) then
        new temp_key:key;
        out(ch_beta, senc((YES, temp_key, xn1_IMD), ECG_based_key))
        out(ch_alpha, penc((YES, temp_key, xn1_Prog), pkProgrammer));

    0.
```

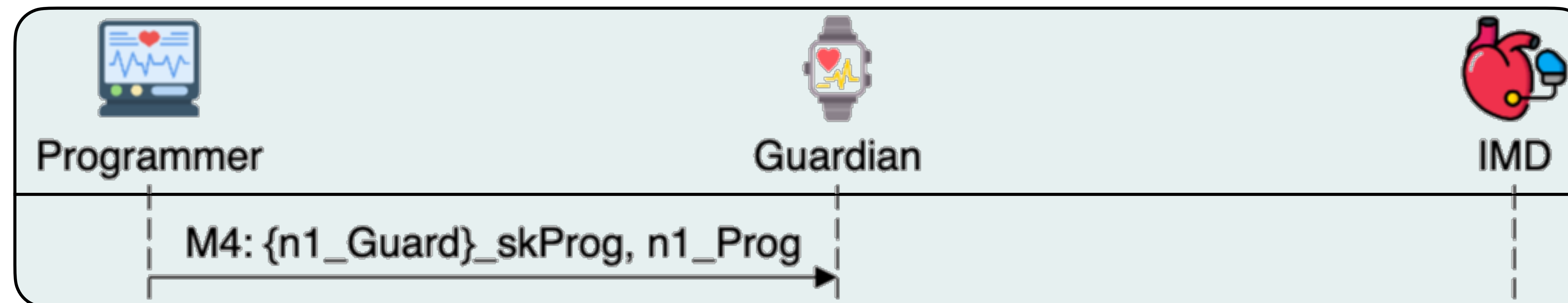
Modeling the Guardian

```
let Guardian(kpGuardian:keymat, ECG_based_key:key, pkProgrammer:pkey) =
    in(ch_beta, (xID_IMD:bitstring, xn1_IMD:bitstring));

    new n1_Guard:bitstring;
    out(ch_alpha, n1_Guard);
    in(ch_alpha, (signature:bitstring, xn1_Prog:bitstring));

    if checksign(signature, pkProgrammer)=ok() then
    if n1_Guard = getmess(signature) then
        new temp_key:key;
        out(ch_beta, senc((YES, temp_key, xn1_IMD), ECG_based_key))
        out(ch_alpha, penc((YES, temp_key, xn1_Prog), pkProgrammer));

    0.
```

Modeling the Guardian

```
let Guardian(kpGuardian:keymat, ECG_based_key:key, pkProgrammer:pkey) =
    in(ch_beta, (xID_IMD:bitstring, xn1_IMD:bitstring));

    new n1_Guard:bitstring;
    out(ch_alpha, n1_Guard);
    in(ch_alpha, (signature:bitstring, xn1_Prog:bitstring));

    if checksign(signature, pkProgrammer)=ok() then
    if n1_Guard = getmess(signature) then
        new temp_key:key;
        out(ch_beta, senc((YES, temp_key, xn1_IMD), ECG_based_key))
        out(ch_alpha, penc((YES, temp_key, xn1_Prog), pkProgrammer));

    0.
```

Modeling the Guardian




```
let Guardian(kpGuardian:keymat, ECG_based_key:key, pkProgrammer:pkey) =
    in(ch_beta, (xID_IMD:bitstring, xn1_IMD:bitstring));

    new n1_Guard:bitstring;
    out(ch_alpha, n1_Guard);
    in(ch_alpha, (signature:bitstring, xn1_Prog:bitstring));

    if checksign(signature, pkProgrammer)=ok() then
    if n1_Guard = getmess(signature) then
        new temp_key:key;
        out(ch_beta, senc((YES, temp_key, xn1_IMD), ECG_based_key))
        out(ch_alpha, penc((YES, temp_key, xn1_Prog), pkProgrammer));

    0.
```

Modeling the Guardian

```
let Guardian(kpGuardian:keymat, ECG_based_key:key, pkProgrammer:pkey) =
    in(ch_beta, (xID_IMD:bitstring, xn1_IMD:bitstring));

    new n1_Guard:bitstring;
    out(ch_alpha, n1_Guard);
    in(ch_alpha, (signature:bitstring, xn1_Prog:bitstring));

    if checksign(signature, pkProgrammer)=ok() then
    if n1_Guard = getmess(signature) then
        new temp_key:key;
        out(ch_beta, senc((YES, temp_key, xn1_IMD), ECG_based_key))
        out(ch_alpha, penc((YES, temp_key, xn1_Prog), pkProgrammer));

    0.
```

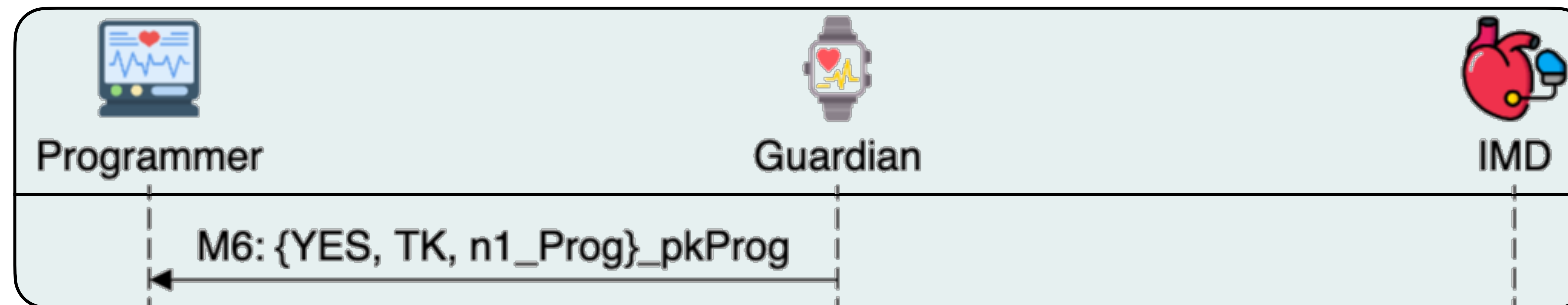
Modeling the Guardian

```
let Guardian(kpGuardian:keymat, ECG_based_key:key, pkProgrammer:pkey) =
    in(ch_beta, (xID_IMD:bitstring, xn1_IMD:bitstring));

    new n1_Guard:bitstring;
    out(ch_alpha, n1_Guard);
    in(ch_alpha, (signature:bitstring, xn1_Prog:bitstring));

    if checksign(signature, pkProgrammer)=ok() then
    if n1_Guard = getmess(signature) then
        new temp_key:key;
        out(ch_beta, senc((YES, temp_key, xn1_IMD), ECG_based_key))
        out(ch_alpha, penc((YES, temp_key, xn1_Prog), pkProgrammer));

    0.
```

Modeling the Guardian

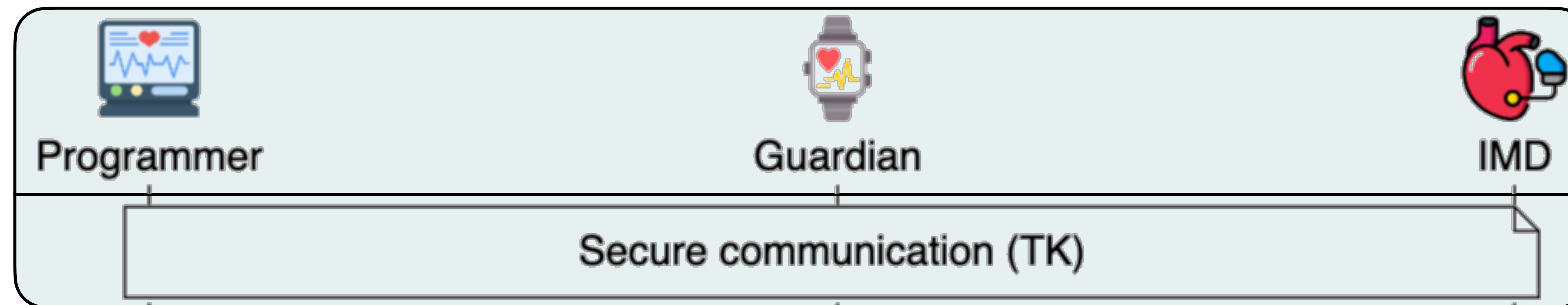
```
let Guardian(kpGuardian:keymat, ECG_based_key:key, pkProgrammer:pkey) =
    in(ch_beta, (xID_IMD:bitstring, xn1_IMD:bitstring));

    new n1_Guard:bitstring;
    out(ch_alpha, n1_Guard);
    in(ch_alpha, (signature:bitstring, xn1_Prog:bitstring));

    if checksign(signature, pkProgrammer)=ok() then
    if n1_Guard = getmess(signature) then
        new temp_key:key;
        out(ch_beta, senc((YES, temp_key, xn1_IMD), ECG_based_key))
        out(ch_alpha, penc((YES, temp_key, xn1_Prog), pkProgrammer));

    0.
```

Modeling the Guardian




```
let Guardian(kpGuardian:keymat, ECG_based_key:key, pkProgrammer:pkey) =
    in(ch_beta, (xID_IMD:bitstring, xn1_IMD:bitstring));

    new n1_Guard:bitstring;
    out(ch_alpha, n1_Guard);
    in(ch_alpha, (signature:bitstring, xn1_Prog:bitstring));

    if checksign(signature, pkProgrammer)=ok() then
    if n1_Guard = getmess(signature) then
        new temp_key:key;
        out(ch_beta, senc((YES, temp_key, xn1_IMD), ECG_based_key))
        out(ch_alpha, penc((YES, temp_key, xn1_Prog), pkProgrammer));

    0.
```

Modeling the Guardian

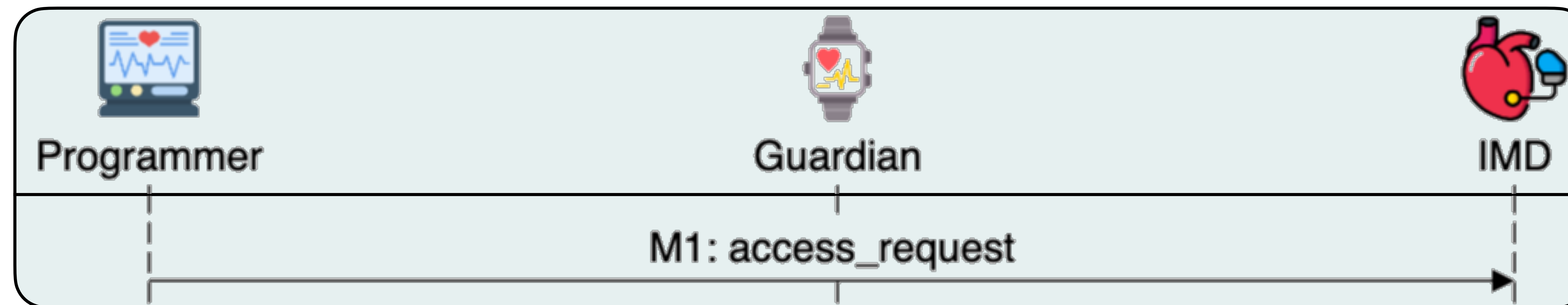



```
let Guardian(kpGuardian:keymat, ECG_based_key:key, pkProgrammer:pkey) =
    in(ch_beta, (xID_IMD:bitstring, xn1_IMD:bitstring));

    new n1_Guard:bitstring;
    out(ch_alpha, n1_Guard);
    in(ch_alpha, (signature:bitstring, xn1_Prog:bitstring));

    if checksign(signature, pkProgrammer)=ok() then
    if n1_Guard = getmess(signature) then
        new temp_key:key;
        out(ch_beta, senc((YES, temp_key, xn1_IMD), ECG_based_key))
        out(ch_alpha, penc((YES, temp_key, xn1_Prog), pkProgrammer));

    0.
```

Modeling the Programmer

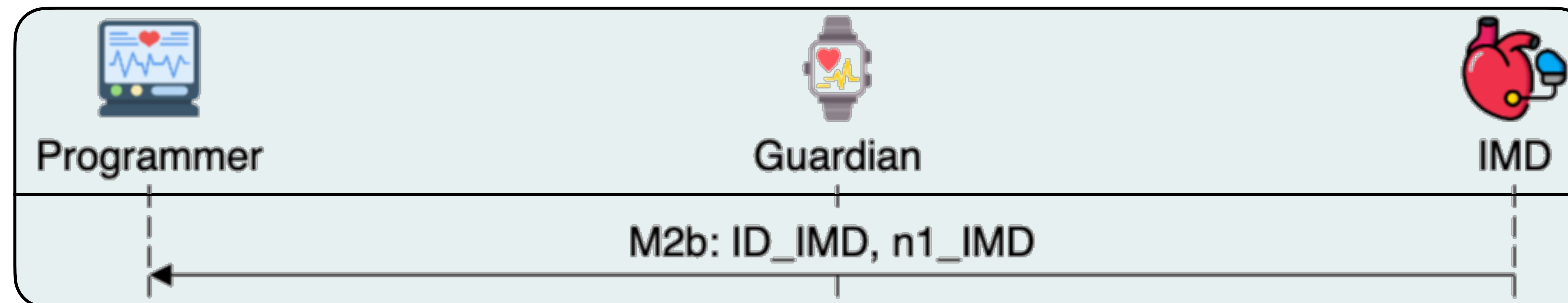
```
let Programmer(kpProgrammer:keymat, pkGuardian:pkey) =
    out(ch_gamma, access_request);
    in(ch_gamma, (xID_IMD:bitstring, xn1_IMD:bitstring));

    in(ch_alpha, xn1_Guard:bitstring);
    new n1_Prog:bitstring;
    out(ch_alpha, (sign(xn1_Guard, sk(kpProgrammer)), n1_Prog));
    in(ch_alpha, ciphertext:bitstring);

    let (=YES, tk:key, =n1_Prog) = pdec(ciphertext, sk(kpProgrammer)) in
        out(ch_gamma, senc(command, tk));
        in(ch_gamma, data:bitstring);
        let xsensitive_data=sdec(data, tk) in
            event phi_received(xsensitive_data, tk)

    0.
```

Modeling the Programmer




```
let Programmer(kpProgrammer:keymat, pkGuardian:pkey) =
    out(ch_gamma, access_request);
    in(ch_gamma, (xID_IMD:bitstring, xn1_IMD:bitstring));

    in(ch_alpha, xn1_Guard:bitstring);
    new n1_Prog:bitstring;
    out(ch_alpha, (sign(xn1_Guard, sk(kpProgrammer)), n1_Prog));
    in(ch_alpha, ciphertext:bitstring);

    let (=YES, tk:key, =n1_Prog) = pdec(ciphertext, sk(kpProgrammer)) in
        out(ch_gamma, senc(command, tk));
        in(ch_gamma, data:bitstring);
        let xsensitive_data=sdec(data, tk) in
            event phi_received(xsensitive_data, tk)

    0.
```

Modeling the Programmer



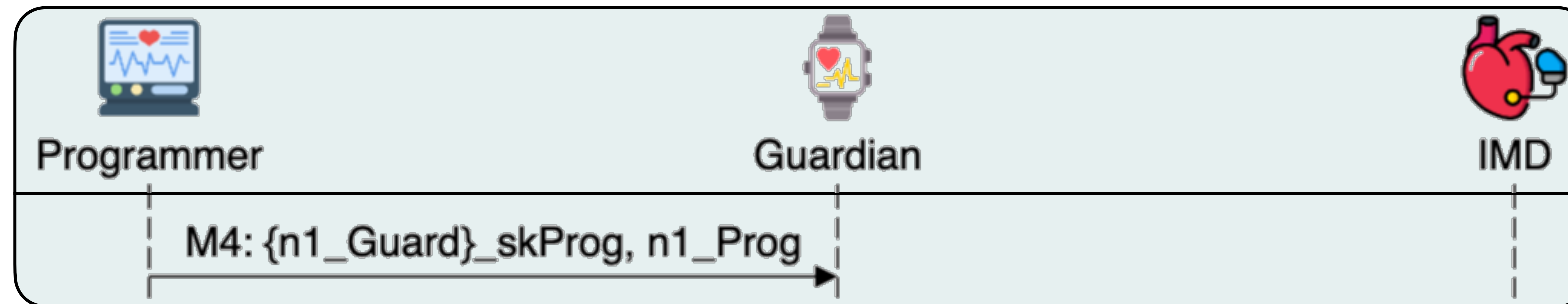

```
let Programmer(kpProgrammer:keymat, pkGuardian:pkey) =
    out(ch_gamma, access_request);
    in(ch_gamma, (xID_IMD:bitstring, xn1_IMD:bitstring));

    in(ch_alpha, xn1_Guard:bitstring);
    new n1_Prog:bitstring;
    out(ch_alpha, (sign(xn1_Guard, sk(kpProgrammer)), n1_Prog));
    in(ch_alpha, ciphertext:bitstring);

    let (=YES, tk:key, =n1_Prog) = pdec(ciphertext, sk(kpProgrammer)) in
        out(ch_gamma, senc(command, tk));
        in(ch_gamma, data:bitstring);
        let xsensitive_data=sdec(data, tk) in
            event phi_received(xsensitive_data, tk)

    0.
```

Modeling the Programmer

```
let Programmer(kpProgrammer:keymat, pkGuardian:pkey) =
    out(ch_gamma, access_request);
    in(ch_gamma, (xID_IMD:bitstring, xn1_IMD:bitstring));

    in(ch_alpha, xn1_Guard:bitstring);
    new n1_Prog:bitstring;
    out(ch_alpha, (sign(xn1_Guard, sk(kpProgrammer)), n1_Prog));
    in(ch_alpha, ciphertext:bitstring);

    let (=YES, tk:key, =n1_Prog) = pdec(ciphertext, sk(kpProgrammer)) in
        out(ch_gamma, senc(command, tk));
        in(ch_gamma, data:bitstring);
        let xsensitive_data=sdec(data, tk) in
            event phi_received(xsensitive_data, tk)

    0.
```

Interacting with the Guardian

Modeling the Programmer

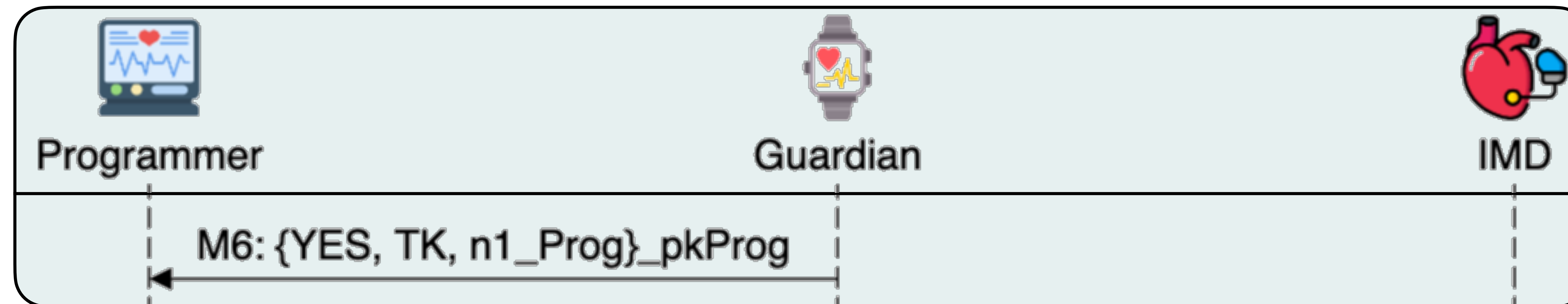



```
let Programmer(kpProgrammer:keymat, pkGuardian:pkey) =
    out(ch_gamma, access_request);
    in(ch_gamma, (xID_IMD:bitstring, xn1_IMD:bitstring));

    in(ch_alpha, xn1_Guard:bitstring);
    new n1_Prog:bitstring;
    out(ch_alpha, (sign(xn1_Guard, sk(kpProgrammer)), n1_Prog));
    in(ch_alpha, ciphertext:bitstring);

    let (=YES, tk:key, =n1_Prog) = pdec(ciphertext, sk(kpProgrammer)) in
        out(ch_gamma, senc(command, tk));
        in(ch_gamma, data:bitstring);
        let xsensitive_data=sdec(data, tk) in
            event phi_received(xsensitive_data, tk)

    0.
```

Modeling the Programmer

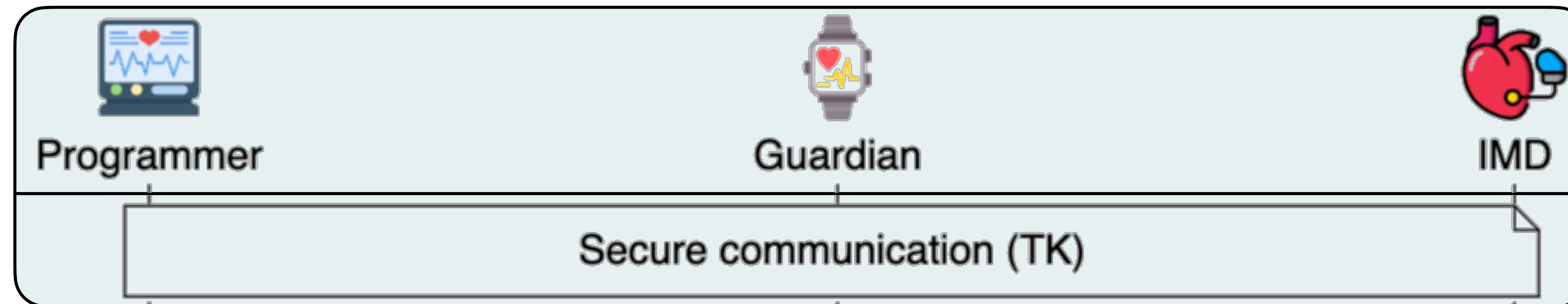
```
let Programmer(kpProgrammer:keymat, pkGuardian:pkey) =
    out(ch_gamma, access_request);
    in(ch_gamma, (xID_IMD:bitstring, xn1_IMD:bitstring));

    in(ch_alpha, xn1_Guard:bitstring);
    new n1_Prog:bitstring;
    out(ch_alpha, (sign(xn1_Guard, sk(kpProgrammer)), n1_Prog));
    in(ch_alpha, ciphertext:bitstring);

    let (=YES, tk:key, =n1_Prog) = pdec(ciphertext, sk(kpProgrammer)) in
        out(ch_gamma, senc(command, tk));
        in(ch_gamma, data:bitstring);
        let xsensitive_data=sdec(data, tk) in
            event phi_received(xsensitive_data, tk)

    0.
```

Modeling the Programmer

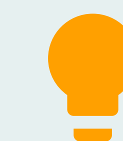



```
let Programmer(kpProgrammer:keymat, pkGuardian:pkey) =
    out(ch_gamma, access_request);
    in(ch_gamma, (xID_IMD:bitstring, xn1_IMD:bitstring));

    in(ch_alpha, xn1_Guard:bitstring);
    new n1_Prog:bitstring;
    out(ch_alpha, (sign(xn1_Guard, sk(kpProgrammer)), n1_Prog));
    in(ch_alpha, ciphertext:bitstring);

    let (=YES, tk:key, =n1_Prog) = pdec(ciphertext, sk(kpProgrammer)) in
        out(ch_gamma, senc(command, tk));
        in(ch_gamma, data:bitstring);
        let xsensitive_data=sdec(data, tk) in
            event phi_received(xsensitive_data, tk)

    0.
```

Modeling the Programmer




```
let Programmer(kpProgrammer:keymat, pkGuardian:pkey) =
    out(ch_gamma, access_request);
    in(ch_gamma, (xID_IMD:bitstring, xn1_IMD:bitstring));

    in(ch_alpha, xn1_Guard:bitstring);
    new n1_Prog:bitstring;
    out(ch_alpha, (sign(xn1_Guard, sk(kpProgrammer)), n1_Prog));
    in(ch_alpha, ciphertext:bitstring);

    let (=YES, tk:key, =n1_Prog) = pdec(ciphertext, sk(kpProgrammer)) in
        out(ch_gamma, senc(command, tk));
        in(ch_gamma, data:bitstring);
        let xsensitive_data=sdec(data, tk) in
            event phi_received(xsensitive_data, tk)

    0.
```

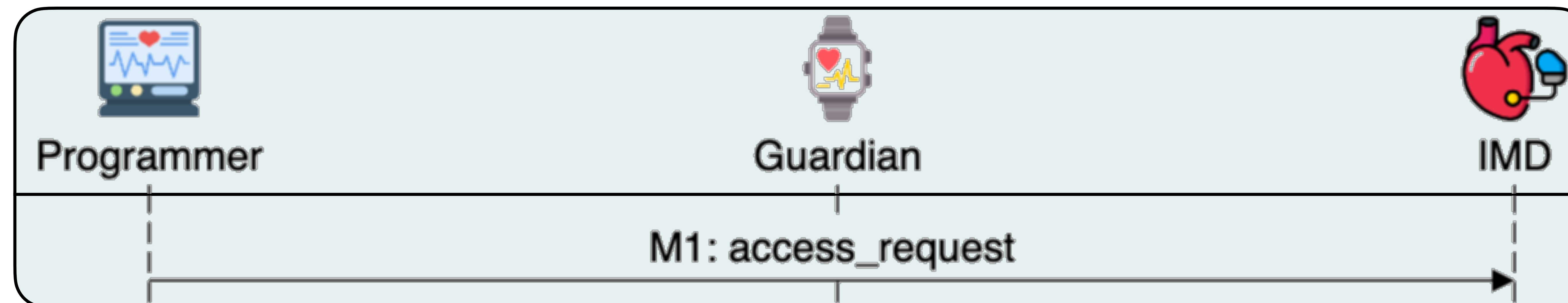


Interacting with the IMD

Modeling the IMD

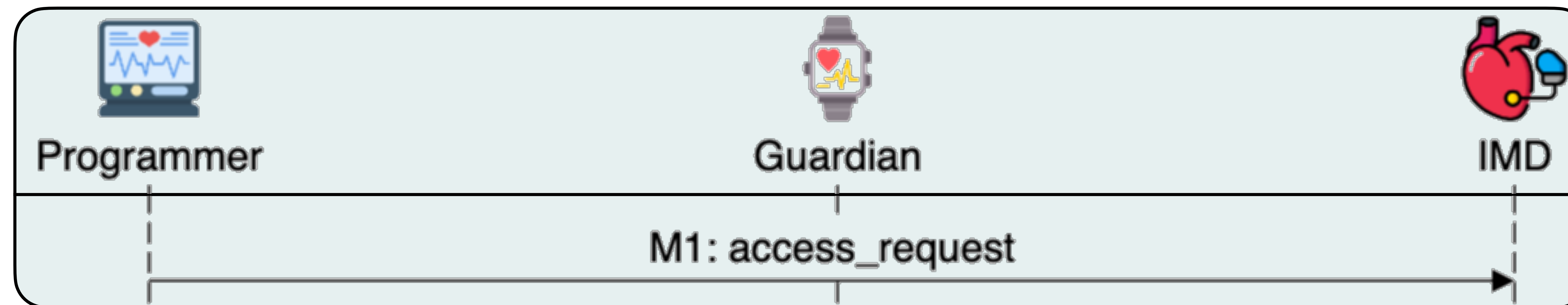
```
let IMD(ECG_based_key:key) =
    in(ch_gamma, request:bitstring);

    new n1_IMD:bitstring;
    out(ch_gamma, (ID_IMD, n1_IMD));
    out(ch_beta, (ID_IMD, n1_IMD));
    in(ch_beta, ciphertext:bitstring);

    let (=YES, tk:key, =n1_IMD) = sdec(ciphertext, ECG_based_key) in
        in(ch_gamma, enc_command:bitstring);
        let xcommand=sdec(enc_command, tk) in
            out(ch_gamma, senc(sensitive_data, tk))
    0.
```

Modeling the IMD




```
let IMD(ECG_based_key:key) =
    in(ch_gamma, request:bitstring);

    new n1_IMD:bitstring;
    out(ch_gamma, (ID_IMD, n1_IMD));
    out(ch_beta, (ID_IMD, n1_IMD));
    in(ch_beta, ciphertext:bitstring);

    let (=YES, tk:key, =n1_IMD) = sdec(ciphertext, ECG_based_key) in
        in(ch_gamma, enc_command:bitstring);
        let xcommand=sdec(enc_command, tk) in
            out(ch_gamma, senc(sensitive_data, tk))
    0.
```

Modeling the IMD

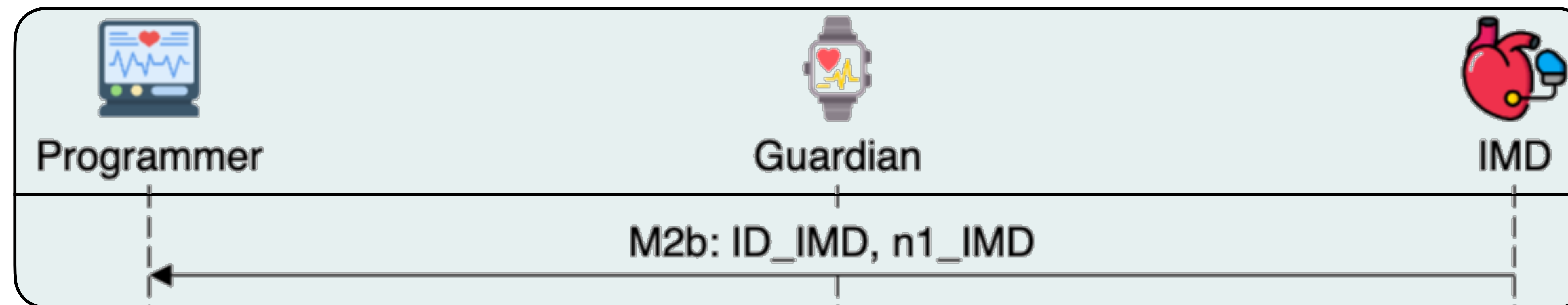



```
let IMD(ECG_based_key:key) =
    in(ch_gamma, request:bitstring);

    new n1_IMD:bitstring;
    out(ch_gamma, (ID_IMD, n1_IMD));
    out(ch_beta, (ID_IMD, n1_IMD));
    in(ch_beta, ciphertext:bitstring);

    let (=YES, tk:key, =n1_IMD) = sdec(ciphertext, ECG_based_key) in
        in(ch_gamma, enc_command:bitstring);
        let xcommand=sdec(enc_command, tk) in
            out(ch_gamma, senc(sensitive_data, tk))
    0.
```

Modeling the IMD

```
let IMD(ECG_based_key:key) =
    in(ch_gamma, request:bitstring);

    new n1_IMD:bitstring;
    out(ch_gamma, (ID_IMD, n1_IMD));
    out(ch_beta, (ID_IMD, n1_IMD));
    in(ch_beta, ciphertext:bitstring);

    let (=YES, tk:key, =n1_IMD) = sdec(ciphertext, ECG_based_key) in
        in(ch_gamma, enc_command:bitstring);
        let xcommand=sdec(enc_command, tk) in
            out(ch_gamma, senc(sensitive_data, tk))
    0.
```

Modeling the IMD




```
let IMD(ECG_based_key:key) =
    in(ch_gamma, request:bitstring);

    new n1_IMD:bitstring;
    out(ch_gamma, (ID_IMD, n1_IMD));
    out(ch_beta, (ID_IMD, n1_IMD));
    in(ch_beta, ciphertext:bitstring);

    let (=YES, tk:key, =n1_IMD) = sdec(ciphertext, ECG_based_key) in
        in(ch_gamma, enc_command:bitstring);
        let xcommand=sdec(enc_command, tk) in
            out(ch_gamma, senc(sensitive_data, tk))
    0.
```

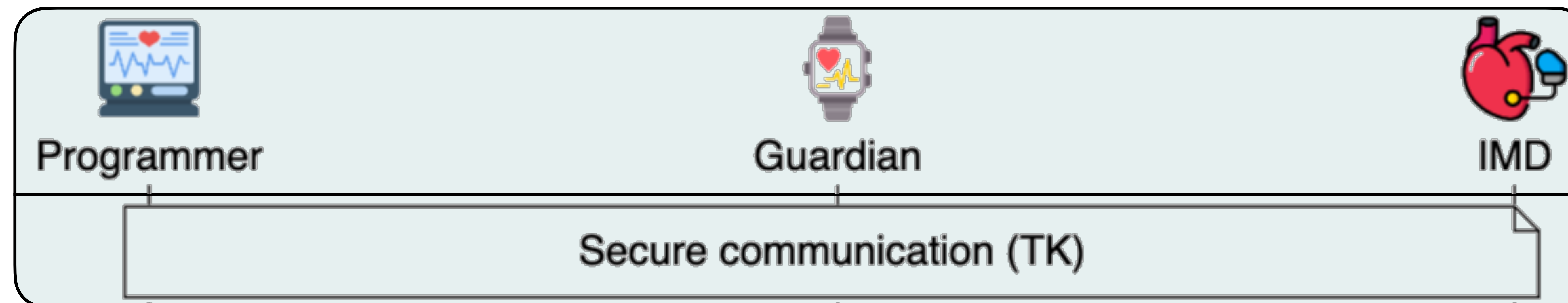
Modeling the IMD

```
let IMD(ECG_based_key:key) =
    in(ch_gamma, request:bitstring);

    new n1_IMD:bitstring;
    out(ch_gamma, (ID_IMD, n1_IMD));
    out(ch_beta, (ID_IMD, n1_IMD));
    in(ch_beta, ciphertext:bitstring);

    let (=YES, tk:key, =n1_IMD) = sdec(ciphertext, ECG_based_key) in
        in(ch_gamma, enc_command:bitstring);
        let xcommand=sdec(enc_command, tk) in
            out(ch_gamma, senc(sensitive_data, tk))
    0.
```

Modeling the IMD

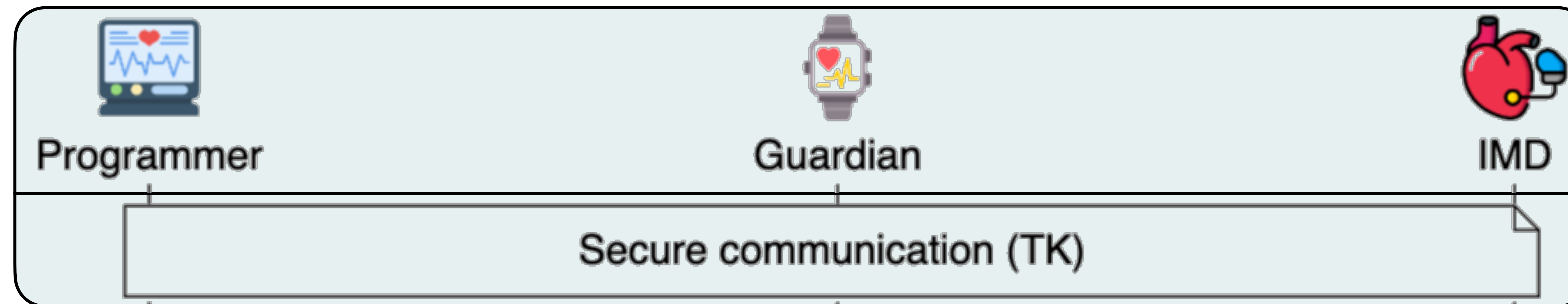
```
let IMD(ECG_based_key:key) =
    in(ch_gamma, request:bitstring);

    new n1_IMD:bitstring;
    out(ch_gamma, (ID_IMD, n1_IMD));
    out(ch_beta, (ID_IMD, n1_IMD));
    in(ch_beta, ciphertext:bitstring);

    let (=YES, tk:key, =n1_IMD) = sdec(ciphertext, ECG_based_key) in
        in(ch_gamma, enc_command:bitstring);
        let xcommand=sdec(enc_command, tk) in
            out(ch_gamma, senc(sensitive_data, tk))
    0.
```

Modeling the IMD




```
let IMD(ECG_based_key:key) =
    in(ch_gamma, request:bitstring);

    new n1_IMD:bitstring;
    out(ch_gamma, (ID_IMD, n1_IMD));
    out(ch_beta, (ID_IMD, n1_IMD));
    in(ch_beta, ciphertext:bitstring);

    let (=YES, tk:key, =n1_IMD) = sdec(ciphertext, ECG_based_key) in
        in(ch_gamma, enc_command:bitstring);
        let xcommand=sdec(enc_command, tk) in
            out(ch_gamma, senc(sensitive_data, tk))
    0.
```

Secure Interaction




```
let Programmer(kpProgrammer:keymat,
               pkGuardian:pkey) =

  (...)

  let (=YES, tk:key, =n1_Prog) =
       pdec(ciphertext, sk(kpProgrammer)) in

    out(ch_gamma, senc(command, tk));
    in(ch_gamma, data:bitstring);
    let xsensitive_data=sdec(data, tk) in
      event phi_received(xsensitive_data, tk)
  0.
```

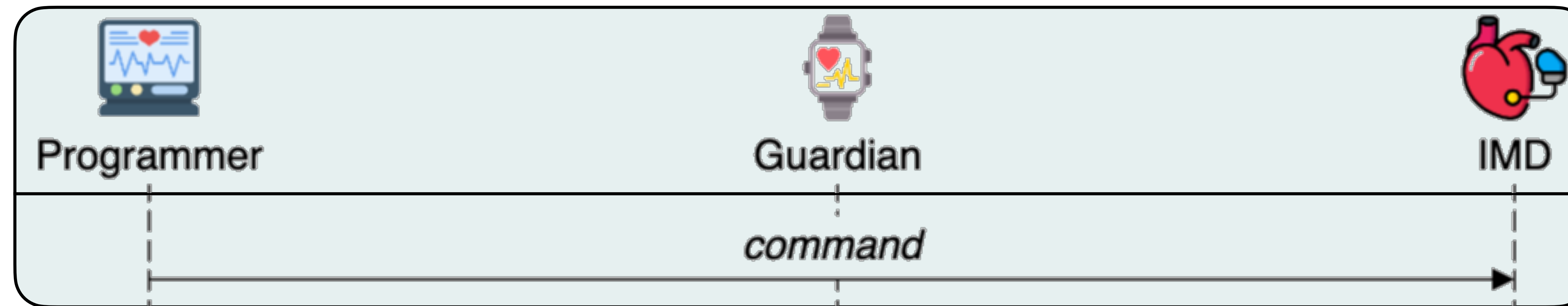
```
let IMD(ECG_based_key:key) =

  (...)

  let (=YES, tk:key, =n1_IMD) =
         sdec(ciphertext, ECG_based_key) in

    in(ch_gamma, enc_command:bitstring);
    let xcommand=sdec(enc_command, tk) in
      out(ch_gamma, senc(sensitive_data, tk))
  0.
```

Secure Interaction

```
let Programmer(kpProgrammer:keymat,
               pkGuardian:pkey) =

  (...)

  let (=YES, tk:key, =n1_Prog) =
      pdec(ciphertext, sk(kpProgrammer)) in

    out(ch_gamma, senc(command, tk));
    in(ch_gamma, data:bitstring);
    let xsensitive_data=sdec(data, tk) in
      event phi_received(xsensitive_data, tk)
  0.
```

```
let IMD(ECG_based_key:key) =

  (...)

  let (=YES, tk:key, =n1_IMD) =
      sdec(ciphertext, ECG_based_key) in

    in(ch_gamma, enc_command:bitstring);
    let xcommand=sdec(enc_command, tk) in
      out(ch_gamma, senc(sensitive_data, tk))
  0.
```

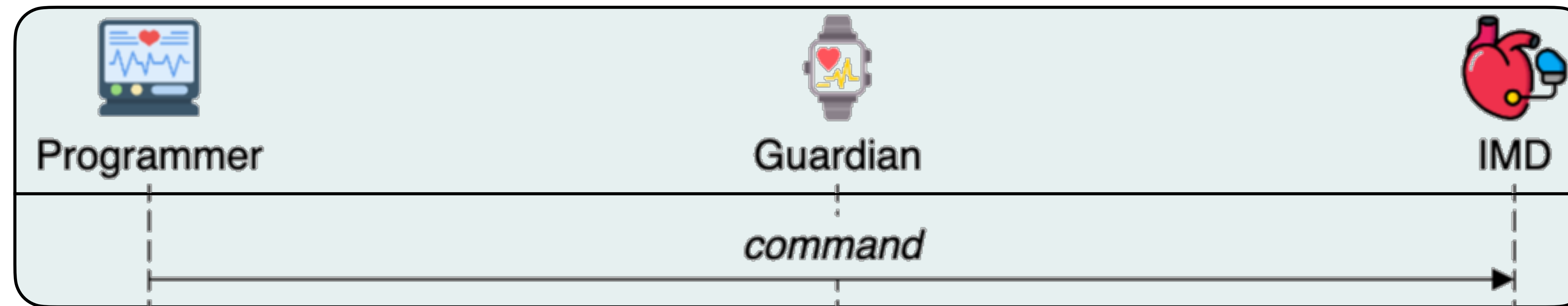
Secure Interaction




```
let Programmer(kpProgrammer:keymat,
               pkGuardian:pkey) =

  (...)

  let (=YES, tk:key, =n1_Prog) =
       pdec(ciphertext, sk(kpProgrammer)) in

    out(ch_gamma, senc(command, tk));
    in(ch_gamma, data:bitstring);
    let xsensitive_data=sdec(data, tk) in
      event phi_received(xsensitive_data, tk)
  0.
```

```
let IMD(ECG_based_key:key) =

  (...)

  let (=YES, tk:key, =n1_IMD) =
        sdec(ciphertext, ECG_based_key) in

    in(ch_gamma, enc_command:bitstring);
    let xcommand=sdec(enc_command, tk) in
      out(ch_gamma, senc(sensitive_data, tk))
  0.
```

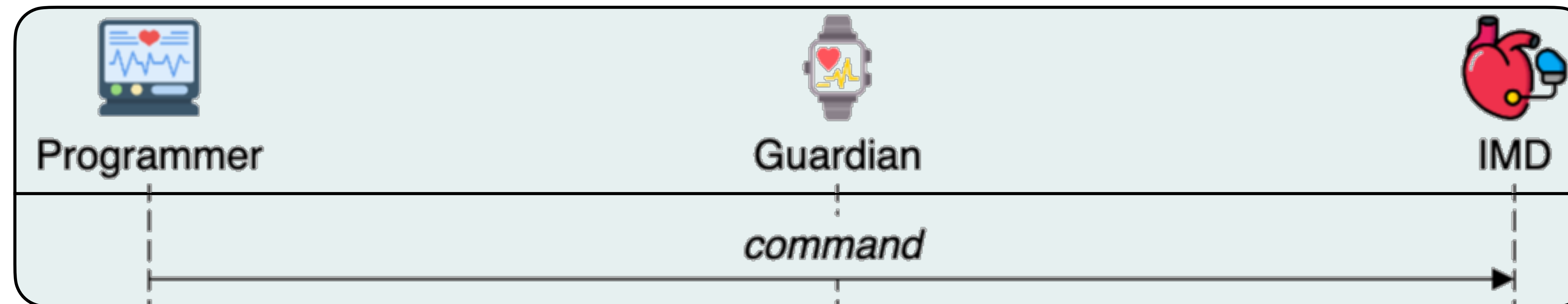
Secure Interaction




```
let Programmer(kpProgrammer:keymat,
               pkGuardian:pkey) =

  (...)

  let (=YES, tk:key, =n1_Prog) =
       pdec(ciphertext, sk(kpProgrammer)) in

    out(ch_gamma, senc(command, tk));
    in(ch_gamma, data:bitstring);
    let xsensitive_data=sdec(data, tk) in
      event phi_received(xsensitive_data, tk)
  0.
```

```
let IMD(ECG_based_key:key) =

  (...)

  let (=YES, tk:key, =n1_IMD) =
        sdec(ciphertext, ECG_based_key) in

    in(ch_gamma, enc_command:bitstring);
    let xcommand=sdec(enc_command, tk) in
      out(ch_gamma, senc(sensitive_data, tk))
  0.
```

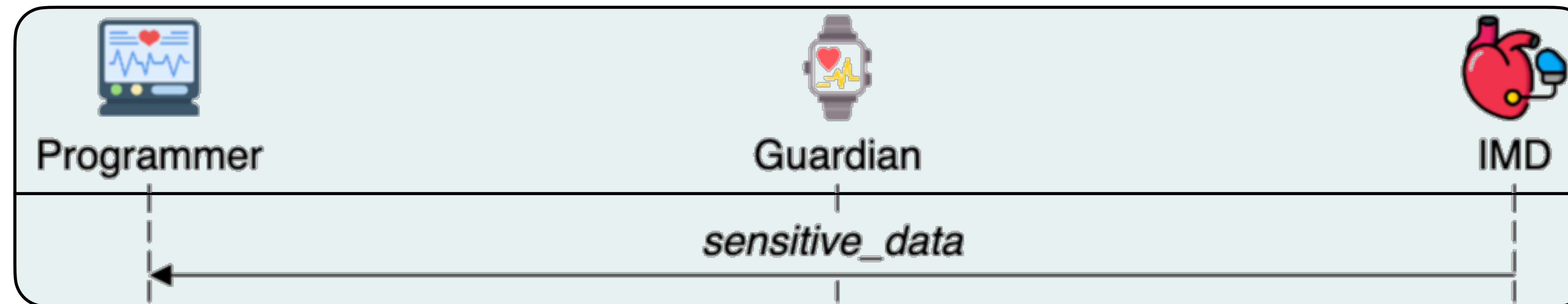
Secure Interaction




```
let Programmer(kpProgrammer:keymat,
               pkGuardian:pkey) =

  (...)

  let (=YES, tk:key, =n1_Prog) =
       pdec(ciphertext, sk(kpProgrammer)) in

    out(ch_gamma, senc(command, tk));
    in(ch_gamma, data:bitstring);
    let xsensitive_data=sdec(data, tk) in
      event phi_received(xsensitive_data, tk)
  0.
```

```
let IMD(ECG_based_key:key) =

  (...)

  let (=YES, tk:key, =n1_IMD) =
        sdec(ciphertext, ECG_based_key) in

    in(ch_gamma, enc_command:bitstring);
    let xcommand=sdec(enc_command, tk) in
      out(ch_gamma, senc(sensitive_data, tk))
  0.
```

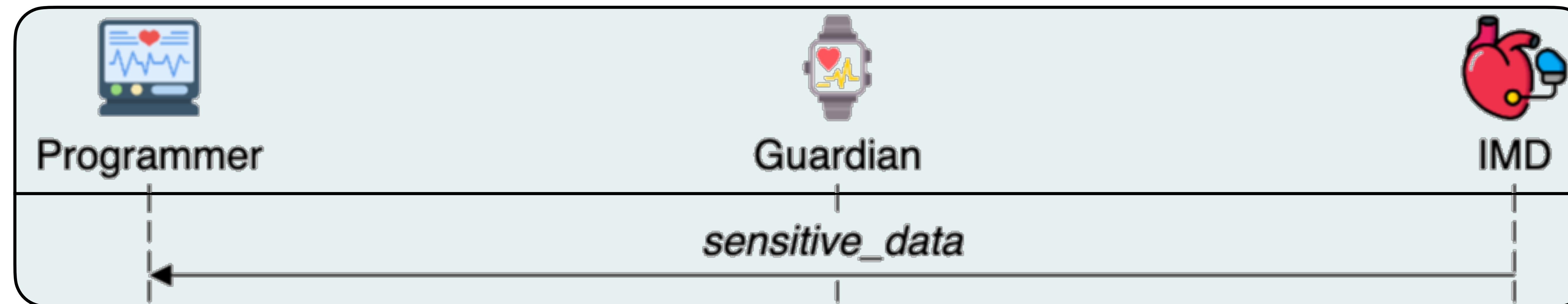
Secure Interaction




```
let Programmer(kpProgrammer:keymat,
               pkGuardian:pkey) =

  (...)

  let (=YES, tk:key, =n1_Prog) =
       pdec(ciphertext, sk(kpProgrammer)) in

    out(ch_gamma, senc(command, tk));
    in(ch_gamma, data:bitstring);
    let xsensitive_data=sdec(data, tk) in
      event phi_received(xsensitive_data, tk)
  0.
```

```
let IMD(ECG_based_key:key) =

  (...)

  let (=YES, tk:key, =n1_IMD) =
        sdec(ciphertext, ECG_based_key) in

    in(ch_gamma, enc_command:bitstring);
    let xcommand=sdec(enc_command, tk) in
     out(ch_gamma, senc(sensitive_data, tk))
  0.
```

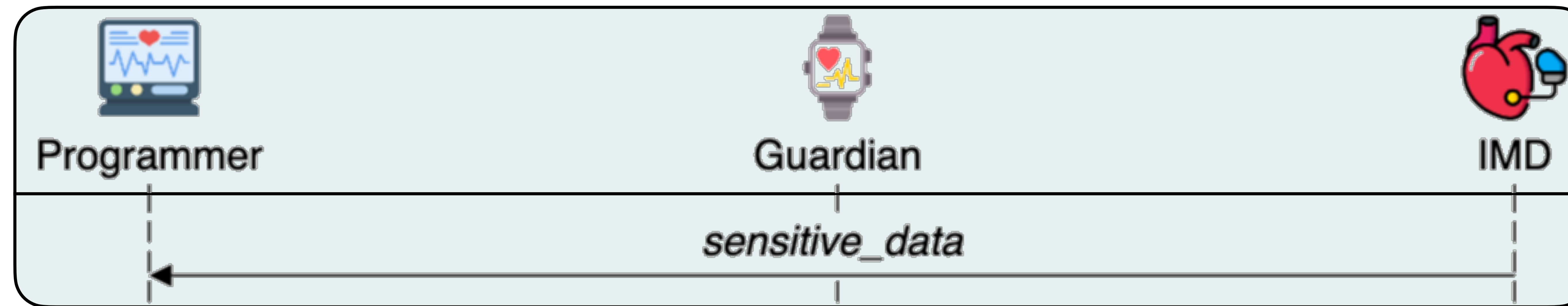
Secure Interaction




```
let Programmer(kpProgrammer:keymat,
               pkGuardian:pkey) =

  (...)

  let (=YES, tk:key, =n1_Prog) =
       pdec(ciphertext, sk(kpProgrammer)) in

    out(ch_gamma, senc(command, tk));
    in(ch_gamma, data:bitstring);
    let xsensitive_data=sdec(data, tk) in
      event phi_received(xsensitive_data, tk)
  0.
```

```
let IMD(ECG_based_key:key) =

  (...)

  let (=YES, tk:key, =n1_IMD) =
        sdec(ciphertext, ECG_based_key) in

    in(ch_gamma, enc_command:bitstring);
    let xcommand=sdec(enc_command, tk) in
     out(ch_gamma, senc(sensitive_data, tk))
  0.
```

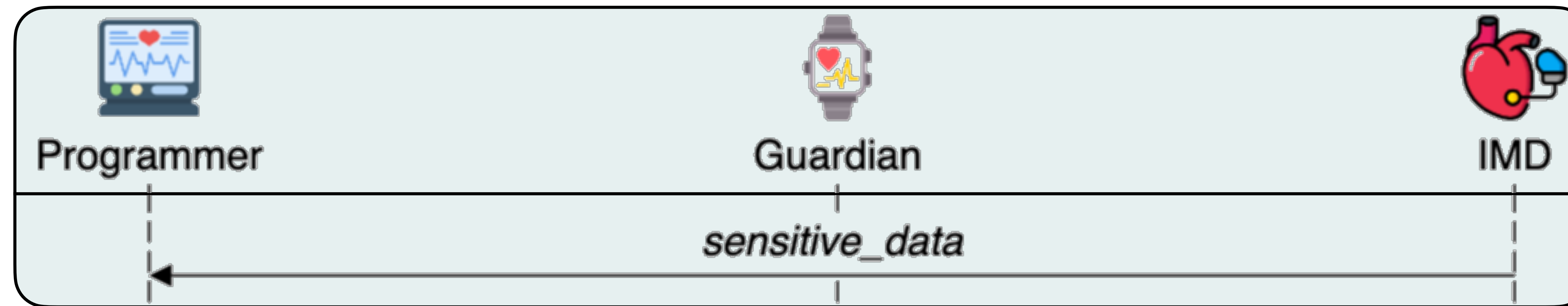
Secure Interaction




```
let Programmer(kpProgrammer:keymat,
               pkGuardian:pkey) =

  (...)

  let (=YES, tk:key, =n1_Prog) =
       pdec(ciphertext, sk(kpProgrammer)) in

    out(ch_gamma, senc(command, tk));
    in(ch_gamma, data:bitstring);
    let xsensitive_data=sdec(data, tk) in
      event phi_received(xsensitive_data, tk)
  0.
```

```
let IMD(ECG_based_key:key) =

  (...)

  let (=YES, tk:key, =n1_IMD) =
        sdec(ciphertext, ECG_based_key) in

    in(ch_gamma, enc_command:bitstring);
    let xcommand=sdec(enc_command, tk) in
     out(ch_gamma, senc(sensitive_data, tk))
  0.
```

Secure Interaction



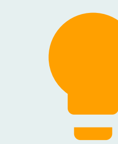

```
let Programmer(kpProgrammer:k
               pkGuardian:pke

  (...)

  let (=YES, tk:key, =n1_Prog
       pdec(ciphertext, sk(kp

    out(ch_gamma, senc(comma
    in(ch_gamma, data:bitstr
    let xsensitive_data=sdec
      event phi_received(xse
  0.
```

```
d_key:key) =

key, =n1_IMD) =
hertext, ECG_based_key) in

 enc_command:bitstring);
=sdec(enc_command, tk) in
ma, senc(sensitive_data, tk))
```

ProVerif Events

*Events are **trace markers** for security-relevant actions that will be verified.*
*They were embedded in the formal model at **critical protocol steps**.*

```
event guardian_allows_comm(pkey, pkey, key).
event programmer_access_allowed(pkey, pkey, key).
event imd_access_allowed(key).
```

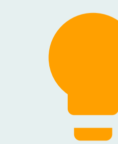


Authorization Events

```
event guardian_denies_comm().
event programmer_access_denied().
event imd_access_denied().
```

Access Denial Events

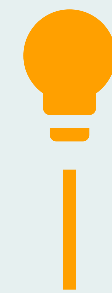
```
event phi_sent(bitstring, key).
event phi_received(bitstring, key).
```



Data Integrity Events

ProVerif Queries

```
query attacker(sensitive_data).
```



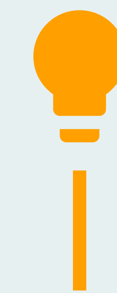
Secrecy queries

Check whether the attacker can derive a secret.

```
query event(guardian_denies_comm()).
query event(programmer_access_denied()).
query event(imd_access_denied()).

query data:bitstring, tk:key;
  inj-event(phi_received(data, tk))
    ⟹ inj-event(phi_sent(data, tk)).

query x:pkey, y:pkey, tk:key;
  inj-event(programmer_access_allowed(x, y, tk))
    ⟹ inj-event(guardian_allows_comm(x, y, tk)).

query x:pkey, y:pkey, tk:key;
  inj-event(imd_access_allowed(tk))
    ⟹ inj-event(guardian_allows_comm(x, y, tk)).
```

ProVerif Queries

```
query attacker(sensitive_data).

query event(guardian_denies_comm()).
query event(programmer_access_denied()).
query event(imd_access_denied()).
```



Reachability queries

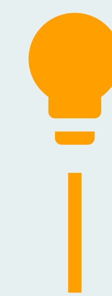
Check whether a specific event can occur.

```
query data:bitstring, tk:key;
  inj-event(phi_received(data, tk))
    ⟹ inj-event(phi_sent(data, tk)).

query x:pkey, y:pkey, tk:key;
  inj-event(programmer_access_allowed(x, y, tk))
    ⟹ inj-event(guardian_allows_comm(x, y, tk)).

query x:pkey, y:pkey, tk:key;
  inj-event(imd_access_allowed(tk))
    ⟹ inj-event(guardian_allows_comm(x, y, tk)).
```

ProVerif Queries

```
query attacker(sensitive_data).

query event(guardian_denies_comm()).
query event(programmer_access_denied()).
query event(imd_access_denied()).
```

```
query data:bitstring, tk:key;
  inj-event(phi_received(data, tk))
    ⟹ inj-event(phi_sent(data, tk)).

query x:pkey, y:pkey, tk:key;
  inj-event(programmer_access_allowed(x, y, tk))
    ⟹ inj-event(guardian_allows_comm(x, y, tk)).

query x:pkey, y:pkey, tk:key;
  inj-event(imd_access_allowed(tk))
    ⟹ inj-event(guardian_allows_comm(x, y, tk)).
```

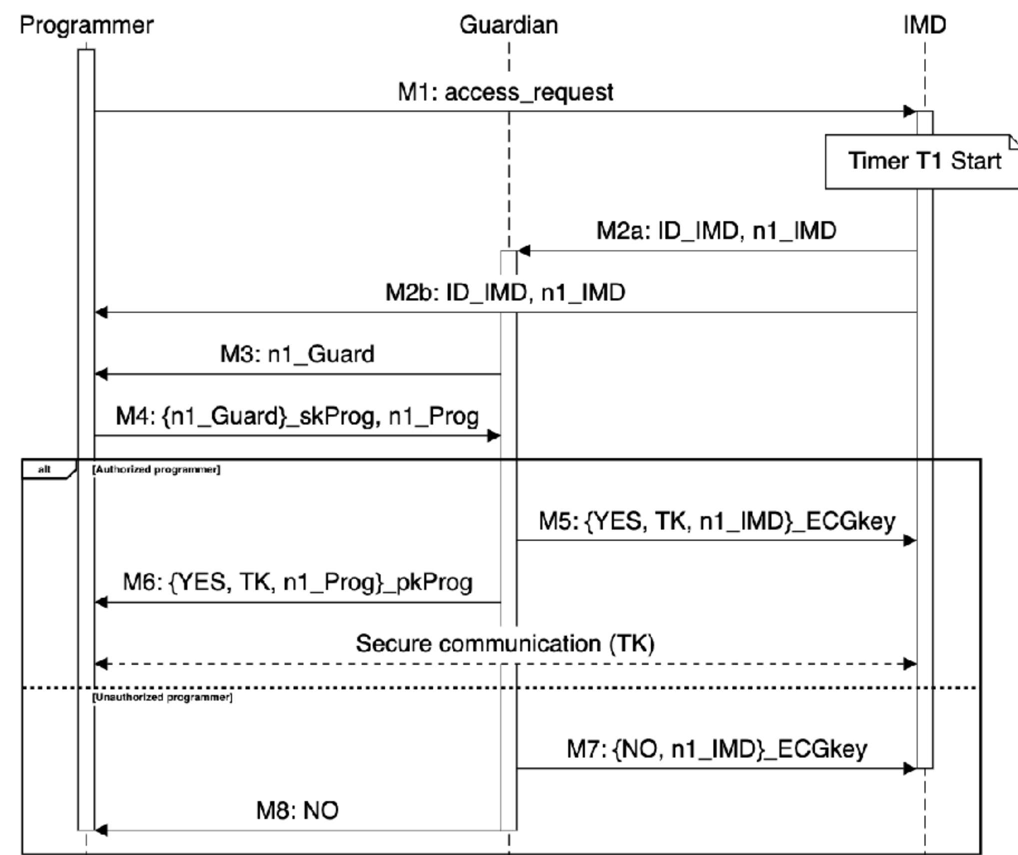


Correspondence queries

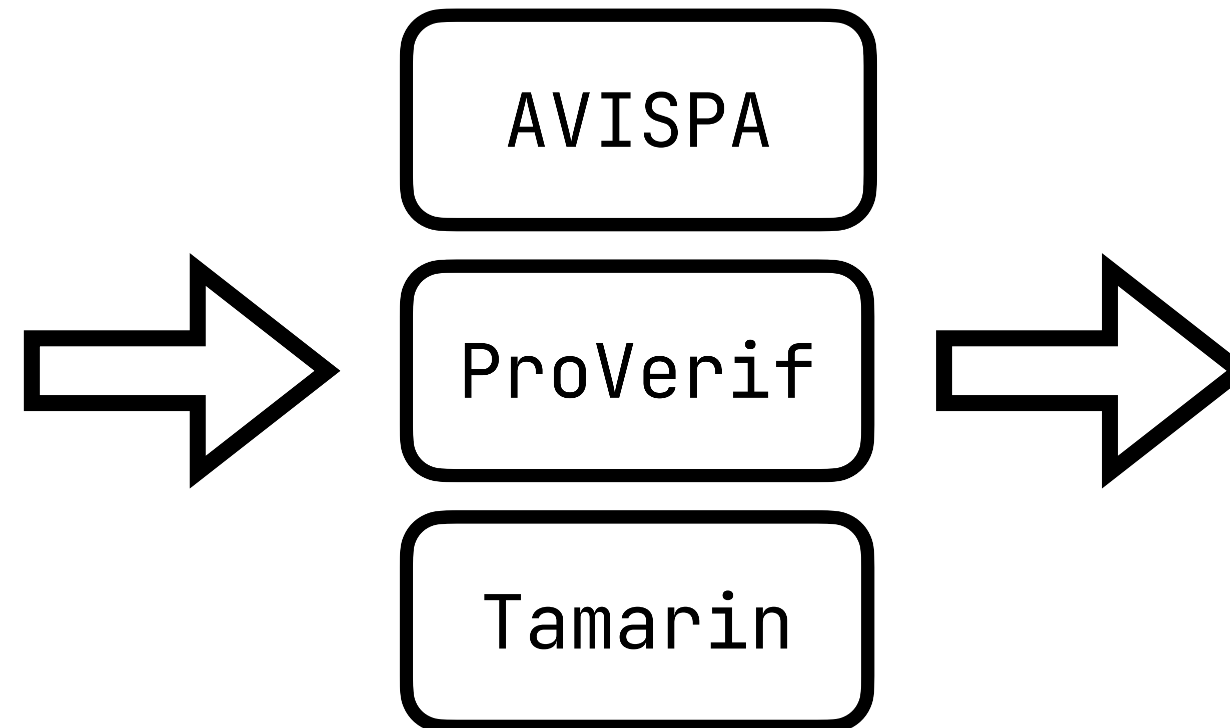
Check that one event occurred before another.

Attack Discovery

Protocol Specifications



Verification Tools



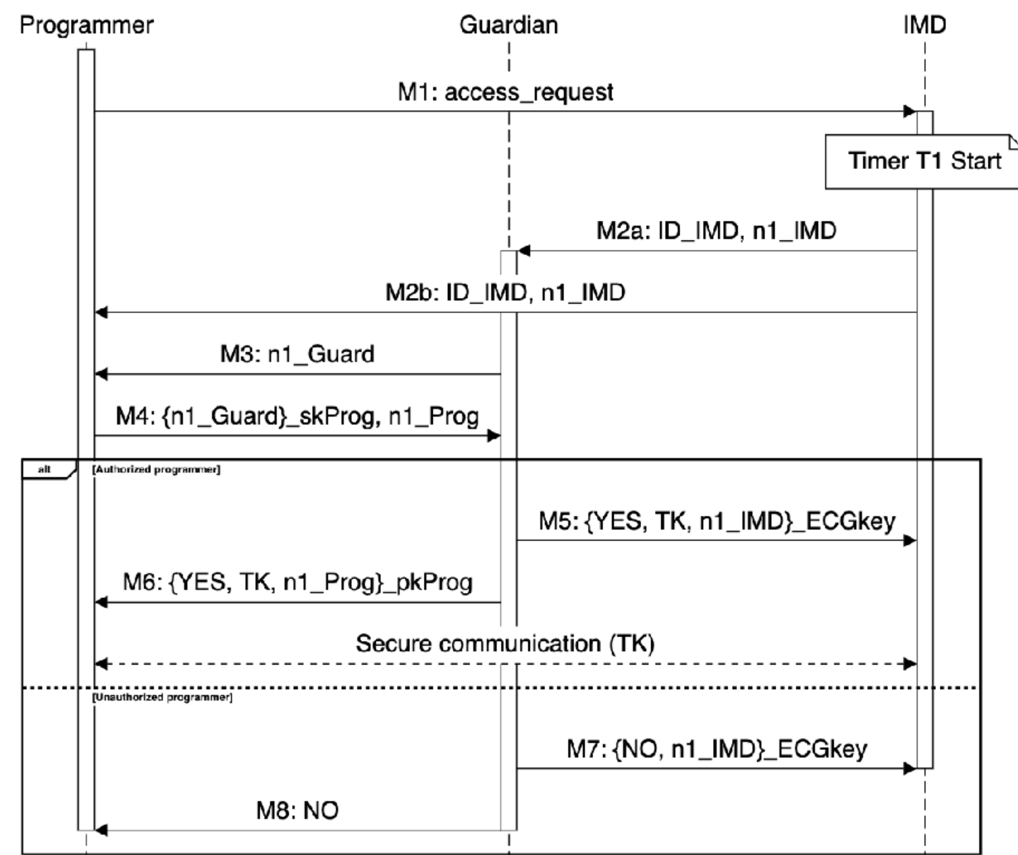
Formal Modeling

```
(* Main process *)
process
    (* Guardian and Programmer have individual public-private key pairs *)
    new kpProgrammer:keymat;
    new kpGuardian:keymat;

    (* The Programmer and IMD share a pre-established EKG-based key *)
    new ECG_based_key:key;

    (
        (* Giving the Public keys to the attacker*)
        out(ch_alpha, pk(kpProgrammer)); 0 | out(ch_alpha, pk(kpGuardian)); 0 |

        (* Instantiating all the processes *)
        IMD(ECG_based_key) |
        Programmer(kpProgrammer, pk(kpGuardian)) |
        Guardian(kpGuardian, ECG_based_key, pk(kpProgrammer))
    )
```

Attack Discovery

Attack Discovery

Protocol Specifications

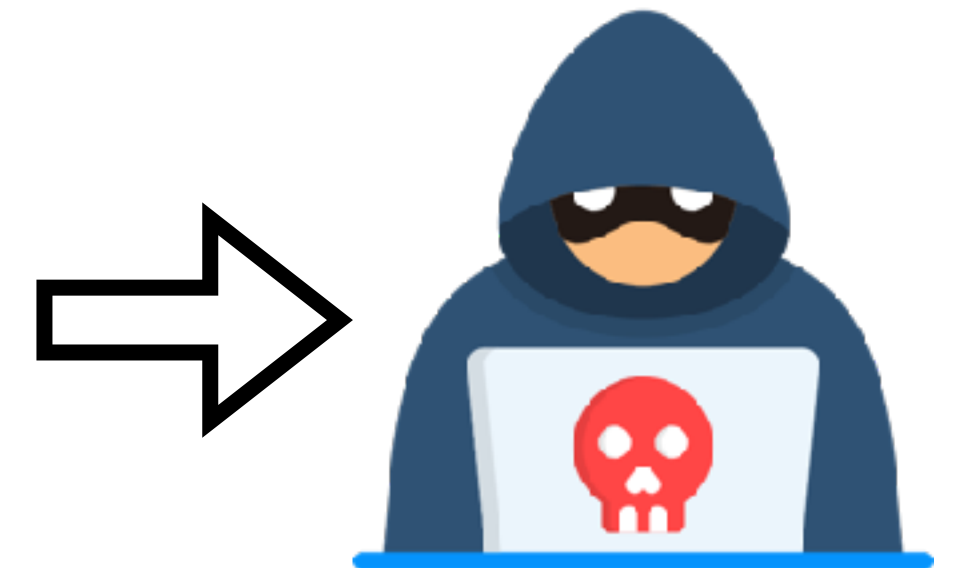


Verification Tools

Formal Modeling

```
(* Main process *)
process
    (* Guardian and Programmer have individual public-private key pairs *)
    new kpProgrammer:keymat;
    new kpGuardian:keymat;

    (* The Programmer and IMD share a pre-established EKG-based key *)
    new ECG_based_key:key;

    (
        (* Giving the Public keys to the attacker*)
        out(ch_alpha, pk(kpProgrammer)); 0 | out(ch_alpha, pk(kpGuardian)); 0 |

        (* Instantiating all the processes *)
        IMD(ECG_based_key) |
        Programmer(kpProgrammer, pk(kpGuardian)) |
        Guardian(kpGuardian, ECG_based_key, pk(kpProgrammer))
    )
```

Attack Discovery



Verification Results

```
Query not attacker(sensitive_data[]) is true.

Query inj-event(imd_access_allowed(tk_2))
  ⟹ inj-event(guardian_allows_comm(x,y,tk_2)) is true.
```

Confidentiality of PHI

Guardian-IMD Authentication

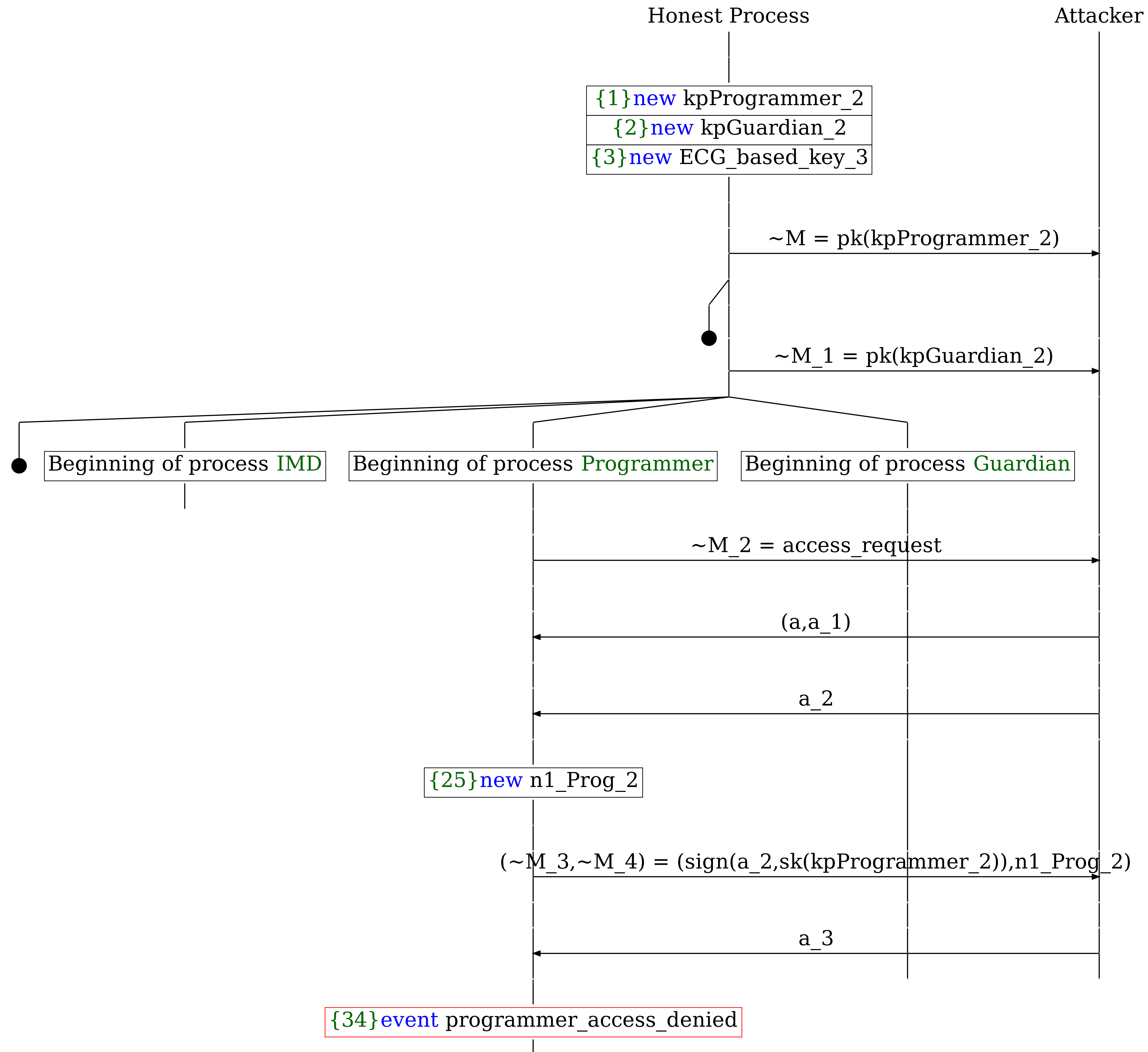
```
Query not event(guardian_denies_comm) is false.
Query not event(programmer_access_denied) is false.
Query not event(imd_access_denied) is false.
```

Reachability of Denial States

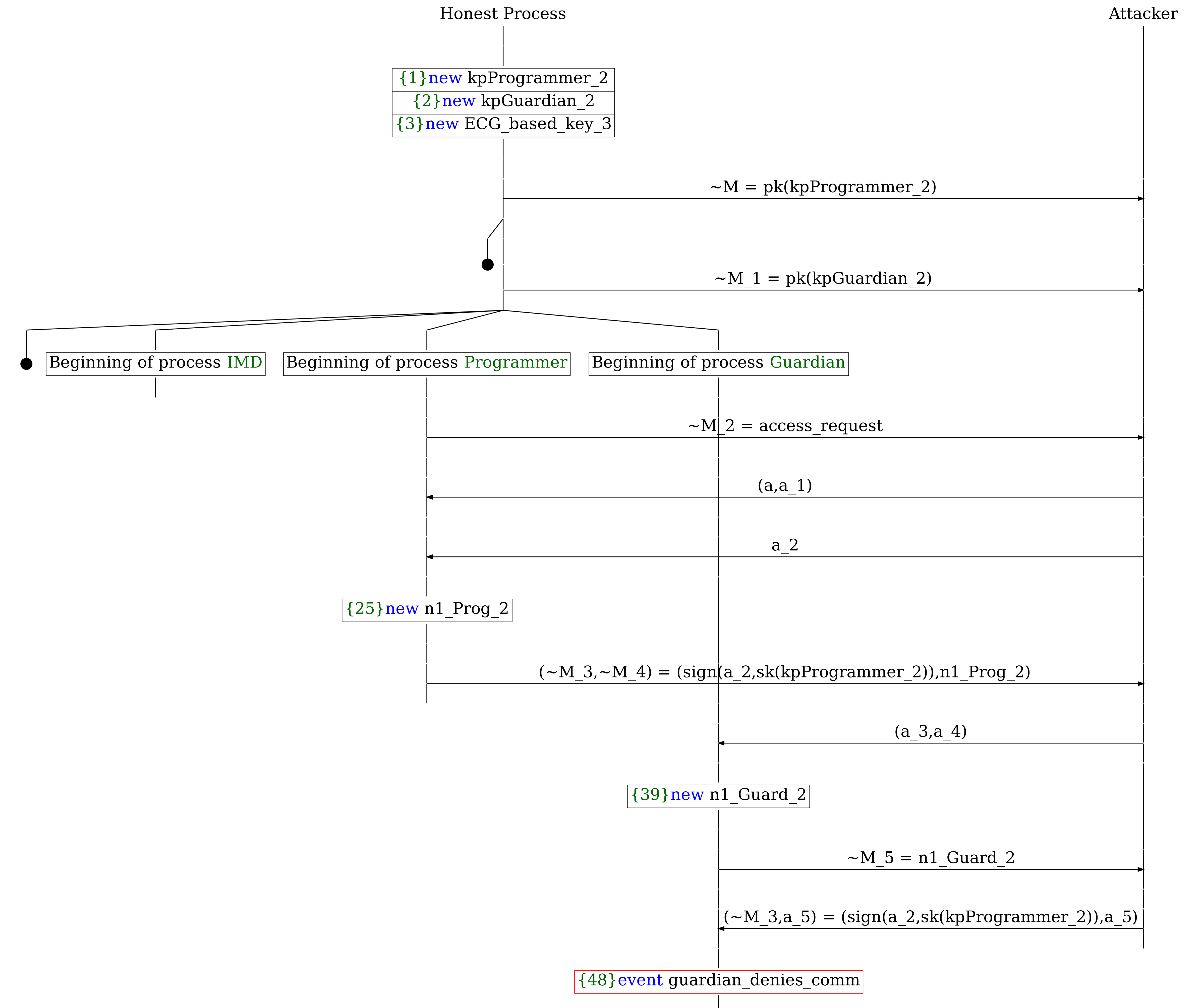
```
Query inj-event(programmer_access_allowed(x,y,tk_2))
  ⟹ inj-event(guardian_allows_comm(x,y,tk_2)) is false.

Query inj-event(phi_received(data_1,tk_2))
  ⟹ inj-event(phi_sent(data_1,tk_2)) is false.
```

Attack Traces

Reachability of *programmer_access_denied*



Reachability of *guardian_denies_comm*



DoS Attacks

> ❗ Wireless **jamming** is the simplest IMD DoS attack, but **not a protocol-level one.**

Programmer

Guardian

IMD

M1: access_request

Timer T1 Start

M2a: ID_IMD, n1_IMD

M2b: ID_IMD, n1_IMD

M3: n1_Guard

Programmer Authentication

alt [Authorized programmer]

M5: {YES, TK, n1_IMD}_ECGkey

M6: {YES, TK, n1_Prog}_pkProg

Secure communication (TK)

[Unauthorized programmer]

M7: {NO, n1_IMD}_ECGkey

M8: NO

DoS Attacks

Three traces in which denial events occur were detected.

Query not event(guardian_denies_comm) is false.

A tampered signature of *n1_Guard* in **M4**.

Programmer
Guardian
IMD
M1: access_request
Timer T1 Start
M2a: ID_IMD, n1_IMD
M2b: ID_IMD, n1_IMD
M3: n1_Guard
M4: {n1_Guard}_skProg, n1_Prog
Programmer Authentication
alt
[Authorized programmer]
M5: {YES, TK, n1_IMD}_ECGkey
M6: {YES, TK, n1_Prog}_pkProg
Secure communication (TK)
[Unauthorized programmer]
M7: {NO, n1_IMD}_ECGkey
M8: NO

DoS Attacks

Three traces in which denial events occur were detected.

```
Query not event(imd_access_denied) is false.
```

Triggered indirectly when the Guardian refuses communication, causing the IMD to also reject the Programmer (**M7**).

Programmer
Guardian
IMD
M1: access_request
Timer T1 Start
M2a: ID_IMD, n1_IMD
M2b: ID_IMD, n1_IMD
M3: n1_Guard
M4: {n1_Guard}_skProg, n1_Prog
Programmer Authentication
alt
[Authorized programmer]
M5: {YES, TK, n1_IMD}_ECGkey
M6: {YES, TK, n1_Prog}_pkProg
Secure communication (TK)
[Unauthorized programmer]
M7: {NO, n1_IMD}_ECGkey
M8: NO

DoS Attacks

Three traces in which denial events occur were detected.

> ❗ `Query not event(programmer_access_denied) is false.`

Forged or modified responses, an incorrect *n1_Prog* in **M6**, lead the Programmer to deny access.

Programmer
Guardian
IMD
M1: access_request
Timer T1 Start
M2a: ID_IMD, n1_IMD
M2b: ID_IMD, n1_IMD
M3: n1_Guard
M4: {n1_Guard}_skProg, n1_Prog
Programmer Authentication
alt
[Authorized programmer]
M5: {YES, TK, n1_IMD}_ECGkey
M6: {YES, TK, n1_Prog}_pkProg
Secure communication (TK)
[Unauthorized programmer]
M7: {NO, n1_IMD}_ECGkey
M8: NO

Verification Results

```
Query not attacker(sensitive_data[]) is true.
```

Confidentiality of PHI

```
Query inj-event(imd_access_allowed(tk_2))
  ⟹ inj-event(guardian_allows_comm(x,y,tk_2)) is true.
```

Guardian-IMD Authentication

```
Query not event(guardian_denies_comm) is false.
Query not event(programmer_access_denied) is false.
Query not event(imd_access_denied) is false.
```

Reachability of Denial States

```
Query inj-event(programmer_access_allowed(x,y,tk_2))
  ⟹ inj-event(guardian_allows_comm(x,y,tk_2)) is false.
```

Guardian Authentication

```
Query inj-event(phi_received(data_1,tk_2))
  ⟹ inj-event(phi_sent(data_1,tk_2)) is false.
```

Forged PHI Acceptance

Attack

- ❌ Guardian Authentication
- ❌ Forged PHI Acceptance

The main flaw is the **absence of mutual authentication.**

Programmer
Guardian
IMD

M1: access_request
Timer T1 Start
M2a: ID_IMD, n1_IMD
M2b: ID_IMD, n1_IMD
M3: n1_Guard
M4: {n1_Guard}_skProg, n1_Prog
Programmer Authentication
alt [Authorized programmer]
M5: {YES, TK, n1_IMD}_ECGkey
M6: {YES, TK, n1_Prog}_pkProg
Secure communication (TK)
[Unauthorized programmer]
M7: {NO, n1_IMD}_ECGkey
M8: NO

Attack: Preconditions

Step 0:

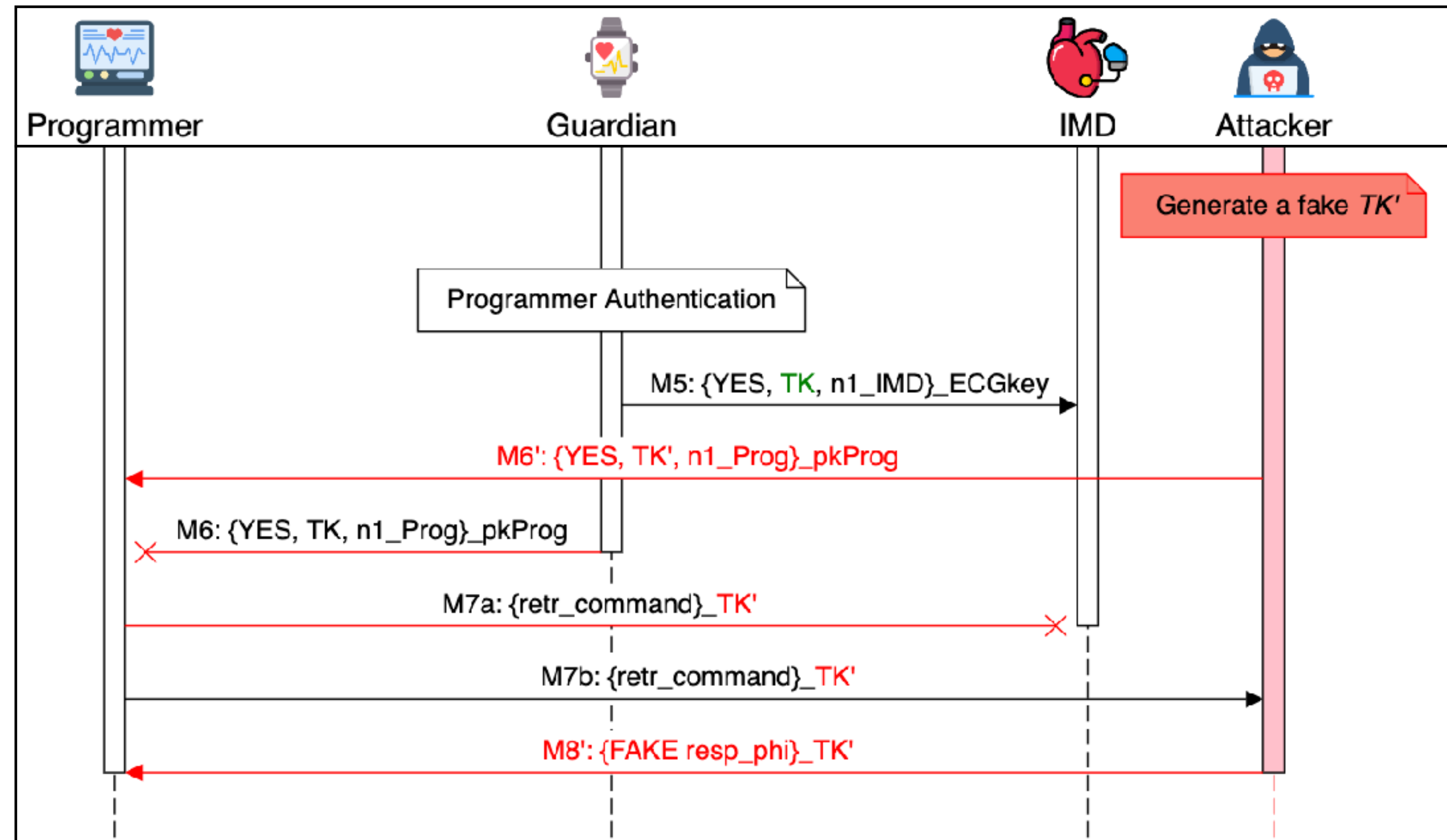
- The attacker knows *pkProg* and the *YES* message.
- Attacker obtains *n1_Prog*.
- The attacker prepares a forged temporary key *TK*'.

Programmer
Guardian
IMD
Attacker
Knows: *pkProg, YES*
M1: access_request
Timer T1 Start
M2a: ID_IMD, n1_IMD
M2b: ID_IMD, n1_IMD
M3: n1_Guard
M4: {n1_Guard}_skProg, n1_Prog
Obtains *n1_Prog*
Generate a fake *TK'*

Attack: Forged TK Installation

Step 1:

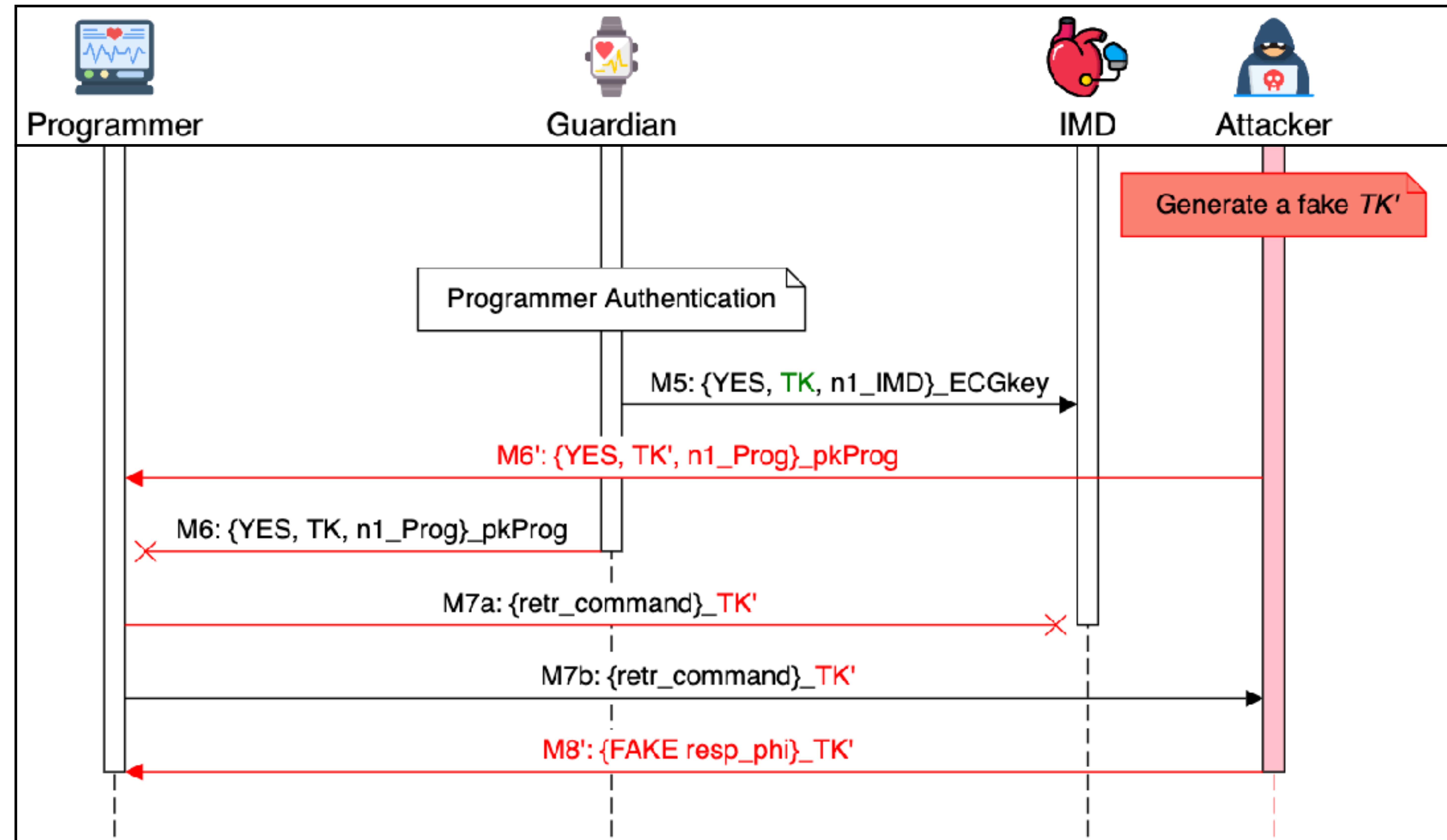
- The attacker forges message *M6'* using the malicious *TK*'.
- The forged response reaches the Programmer before the legitimate Guardian message.
- The Programmer accepts *TK*' as a valid session key.



Attack: Fake Data Injection

Step 2:

- Using the *TK*', the attacker impersonates the IMD.
- The Programmer sends encrypted commands using the malicious session key.
- The attacker responds with counterfeit medical data.



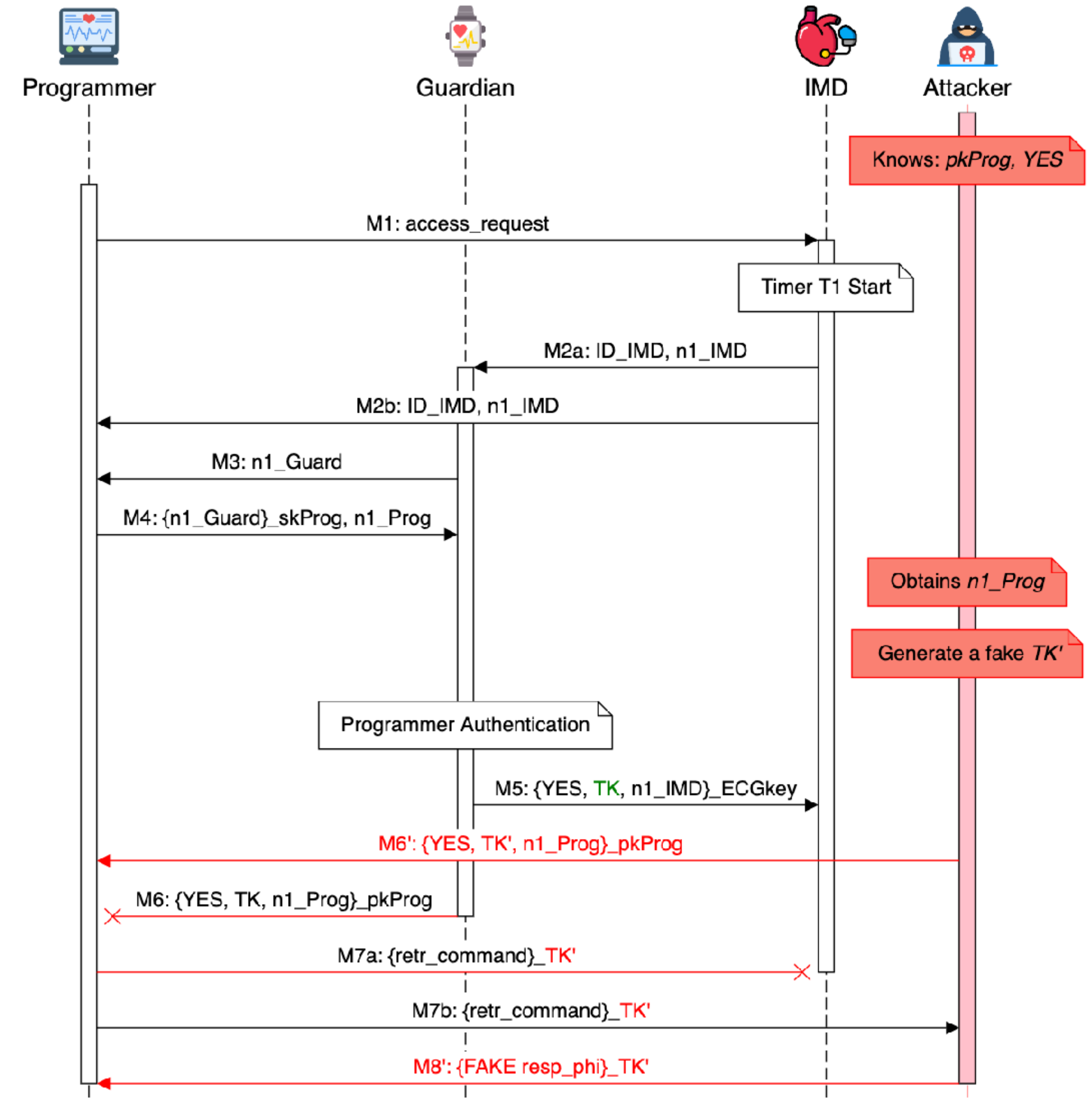
Attack: Queries Explanation

Guardian Authentication

```
inj-event(programmer_access_allowed(x,y,tk_2)) ⟹
  inj-event(guardian_allows_comm(x,y,tk_2))
```

Forged PHI Acceptance

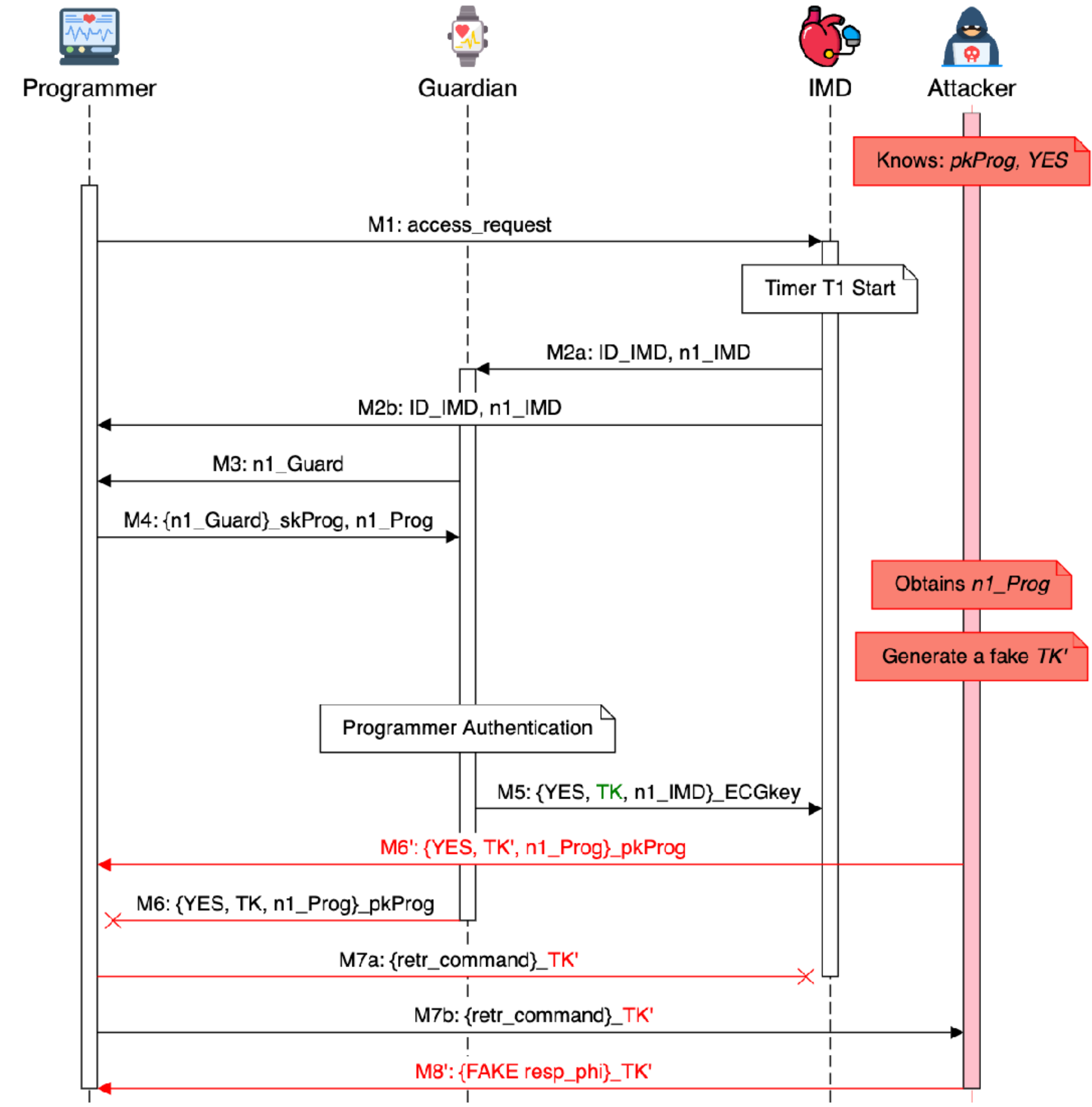
```
inj-event(phi_received(data_1,tk_2)) ⟹
  inj-event(phi_sent(data_1,tk_2))
```



Attack: Impact

The attacker can make the Programmer accept forged IMD data.

- **Limitation**: the attacker cannot impersonate the Programmer or inject unauthorized IMD commands.
- **Security Risk**: manipulated telemetry may mislead physicians and indirectly affect patient safety.



Fixing IMDGuard

Programmer
Guardian
IMD

M1: access_request
Timer T1 Start
M2a: ID_IMD, n1_IMD
M2b: ID_IMD, n1_IMD
M3: n1_Guard
M4: {n1_Guard}_skProg, n1_Prog
Programmer Authentication
M5: {YES, TK, n1_IMD}_ECGkey
M6: {{YES, TK, n1_Prog}_skGuard}_pkProg
Guardian Authentication
alt [Authorized Programmer and Guardian]
Secure communication (TK)

Ensure the temporary key is sent by the legitimate Guardian, **enabling mutual authentication**.

If you want to read more…

TELMED 2026: Fifth International Workshop on Telemedicine and e-Health in the digital society

Formal Verification and Mitigation of Vulnerabilities in the IMDGuard Protocol

Christian Coduri, Alessio Sacco, Guido Marchetto, Riccardo Sisto
Department of Control and Computer Engineering, Politecnico di Torino, Turin, Italy.
Emails: alessio_sacco@polito.it, {first.last}@polito.it

Abstract*—IMDGuard is a proxy-based authentication protocol designed to secure communications between an Implantable Medical Device (IMD), a technology increasingly used in cardiac care, and its associated Programmer device through a wearable third-party mediator known as the Guardian. In this paper, we formally analyze the security of IMDGuard using the ProVerif verification tool and demonstrate that the protocol is vulnerable to multiple attacks. Our analysis confirms a known susceptibility to denial-of-service attacks and reveals a previously unreported flaw in the authentication mechanism between the Guardian and the Programmer. This vulnerability allows an adversary to impersonate the IMD, inject falsified telemetry data, and issue deceptive responses to legitimate Programmer commands, posing a serious risk to patient safety by potentially leading clinicians to make medical decisions based on compromised information. To mitigate this threat, we propose a remediation mechanism that enforces the mutual authentication process, thereby enhancing protocol resilience against impersonation attacks. The

access to the IMD to retrieve data or modify therapy parameters while the patient is unable to provide the required credentials (e.g., due to unconsciousness). In such scenarios, strict authentication and access-control mechanisms may be impractical [3]. Therefore, two access modalities are required: a method for normal operation and an emergency mode that temporarily bypasses security to permit what would otherwise be considered unauthorized access.

To secure communication with IMDs, various authentication and access control protocols have been proposed [4], operating either in both normal and emergency modes or exclusively in normal mode, and relying on proximity, biometrics, or proxy-based mechanisms. Proxy-based approaches are particularly promising for real-world deployment, as they offload computationally intensive cryptographic operations to an external

Conclusions

“In the end, what matters is not only whether the math holds, but whether the logic truly does.”

- Formal verification helps identify **subtle flaws** before deployment
- Modern **IMDs** are connected cyber-physical systems whose security protocols require **rigorous analysis**.
- Formal analysis should be considered for both **existing and future security protocols**, not only in the medical domain but **across systems in general**.

Thank You!

Christian Coduri
christian.coduri@polito.it