The Journal of Systems and Software 230 (2025) 112501

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Correctness and security analysis of the protection in transit (PIT) protocol ☆

Rakesh Podder [a,*], Mahmoud Abdelgawad [a], Indrakshi Ray [a], Indrajit Ray [a,*], Madhan Santharam [b], Stefano Righi [b]

[a] Department of Computer Science, Colorado State University, Fort Collins, 80523, CO, USA
[b] AMI US Holdings Inc., Duluth, 30096, GA, USA

ARTICLE INFO

Dataset link: https://github.com/cryptoknight13/Project-Cerberus

Keywords:
Firmware security
BIOS
Hardware root
Secure boot
STRIDE
Formal methods
Coloured Petri Nets (CPN)

ABSTRACT

A computing device's firmware, such as the Basic Input Output System (BIOS), is a critical component for the device's operation since it is the software that allows a device's hardware to communicate with the operating system and must be protected from unauthorized and malicious modifications. Such modifications are serious, since control over the firmware allows an attacker to fully control the entire computing stack, bypassing all security controls provided by the operating system and upper layers. Although the industry has developed protocols to secure the boot process, once a device boots up, it is easy to tamper with the firmware. This threat may occur when computing devices are shipped and can be booted up by malicious attackers who can intercept the shipment. We developed the Protection in Transit (PIT) protocol that mitigates this threat by locking a device during transit and ensuring that the device can be booted up only by an authorized user. We provide formal guarantees about the correctness and security of the PIT protocol using Coloured Petri Nets (CPN) and demonstrate robustness against potential cyberattacks that we enumerate with the Microsoft STRIDE threat modeling framework. We also discuss the implementation of the PIT protocol that consists of libraries using a cryptographic technique that have been rigorously tested on a trusted embedded framework. Our results demonstrate that the PIT protocol is robust and can be deployed to prevent firmware tampering during transit.

1. Introduction

Firmware is the essential software that allows the hardware of a device to communicate with the operating system (Ganssle, 2004). When a computer is turned on, the Basic Input Output System (BIOS) firmware initializes the hardware by loading the System Management Interrupts (SMIs), starting the Advanced Configuration and Power Interface (ACPI), and initiating the loading of the operating system (OS). The firmware operates in a high-privilege CPU mode, distinct from the standard operating system execution mode. Unified Extensible Firmware Interface (UEFI) that replaces the BIOS and Baseboard Management Controllers (BMC) firmware used in cloud servers behaves similarly. We use the term firmware or BIOS to mean all three of these technologies. Any compromise to the BIOS gives an attacker complete control over a device. Furthermore, once this device boots up, the attacker can compromise other components. Consequently, firmware must be protected from unauthorized tampering.

Various technologies, such as Secure Boot (Löhr et al., 2010), Trusted Boot (Khalid et al., 2013), Remote Attestation (Coker et al., 2011), and the NIST Special Publication 800-193 (Regenscheid, 2018), try to provide mechanisms to secure device booting during system initialization and operation. These works assume that the BIOS has not been tampered with. These methodologies do not protect against the tampering of the BIOS itself. Thus, an attacker with access to the device can initiate a boot process, then interrupt it before the full BIOS is loaded, and replace the good BIOS with a malicious one. (The attack is described in Section 7). This threat is possible when a device is shipped from one site to another; for example, from a vendor to a customer or from one branch of an organization to another.

The need to protect the device firmware in transit led to our work (Podder et al., 2024), which ensures that only an authorized user is allowed access to boot the device. We achieve this goal through an innovative approach, which we call the Protection in Transit Protocol (PIT Protocol). In this protocol, the device manufacturer implements a BIOS lock post-production and introduces a mechanism for user authentication before unlocking it. The device is attached to a separate trusted microcontroller (Gui et al., 2018), which acts as a Hardware

☆ Editor: W. Eric Wong.

* Corresponding authors.
E-mail addresses: rakesh.podder@colostate.edu (R. Podder), indrajit.ray@colostate.edu (I. Ray).

https://doi.org/10.1016/j.jss.2025.112501
Received 30 September 2024; Received in revised form 29 March 2025; Accepted 19 May 2025
Available online 5 June 2025

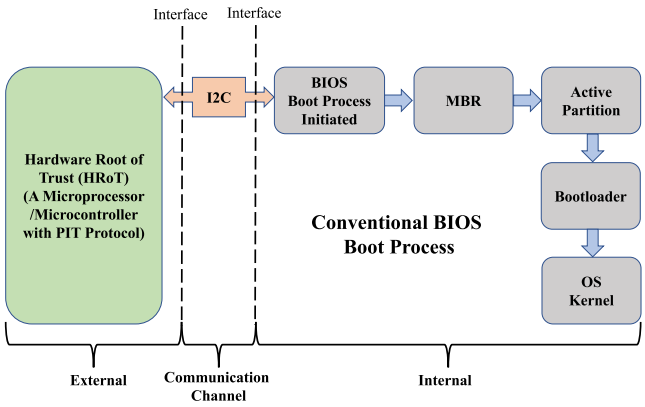

Fig. 1. HRoT Precedes Boot Process of BIOS.

Root of Trust (*HRoT*) for reliable machine boot. The *HRoT* may be a separate chip on the motherboard. We use the term product to refer to the physical object that embeds the *HRoT* and the device, which is shipped to *Recipient*.

The *HRoT* role in the BIOS boot process is to authenticate the device *Recipient* and delay the traditional BIOS boot process until this authentication has been completed. It does this using a lock and an unlock mechanism. *Locking* prevents the initialization of BIOS boot sequences through the traditional "power on/off" method. *Unlocking* occurs when the boot sequences start and the OS is loaded into the kernel. The device lock operation is executed in a BIOS and the *HRoT* triggers the unlock operation after successful authentication.

We implemented the PIT protocol on a trusted embedded framework that serves as a *HRoT* and integrated it into the BIOS boot process. Fig. 1 illustrates how *HRoT*, acting as a gatekeeper, communicates with the BIOS via the Inter-Integrated Circuit (I2C) protocol, secures the BIOS and verifies the *Recipient*'s credentials before the BIOS is allowed to boot up. If the *Recipient* is the rightful owner of the product, the *Recipient* is granted access to the product, and the BIOS initializes the hardware. Then, the BIOS calls the code stored in the Master Boot Record (MBR) at the start of disk 0. The MBR loads code from the bootsector of the active partition. The bootsector loads and runs the bootloader from the filesystem. Finally, the bootloader locates and loads the kernel for the selected OS from the disk and gives control of the PC to the OS. Upon successful completion of this process, the product becomes accessible and usable by *the Recipient*.

Since the PIT protocol plays an important role in the overall security of the device, ensuring correct execution of the protocol and its security is critical. The correct execution of the PIT protocol is equivalent to ensuring that the device *Recipient* and the device mutually authenticate each other.

We develop a methodology to verify the PIT protocol to provide the correctness and security assurance. We first represent the PIT protocol using Unified Modeling Language (UML) sequence diagrams (Ordinez et al., 2020) to describe message exchange between the entities of the PIT protocol. The primary motivation for using UML is that it is the de facto specification language used in the software industry. However, few tools are available for automated analysis of UML diagrams. Toward this end, we use Coloured Petri Nets (CPN), a graphical language based on mathematical theory used for modeling and validating concurrent and distributed systems (Jensen and Kristensen, 2015; Jensen et al., 2007), to analyze our PIT protocol. CPN is suitable for modeling process interaction and provides primitives for defining data types. It is associated with a high-level programming language, CPN-ML (Jensen and Kristensen, 2009), and an integrated development environment (i.e., CPN Tools) that are practical for simulating process interaction, manipulating data values, and verifying state transitions of systems. We provide algorithms to convert UML sequence diagrams to CPN. Once the CPN has been analyzed to provide the assurance of correctness, we focus on the security aspects.

We use a threat modeling framework, Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege (STRIDE) (Shostack, 2014), to evaluate threats in the PIT protocol. The Microsoft Threat Modeling Tool (MTMT) (Shostack, 2014) is used to generate threat models based on Data Flow Diagram (DFD), which is a diagramming technique to describe the data flow between system entities and processes (Li and Chen, 2009). The DFD presents the process flow of the PIT protocol and classifies threats into the categories defined in STRIDE. These threats are converted to CPN attacks and integrated into the CPN of the PIT protocol. This integration helps to analyze the impact of various cyberattack scenarios on the PIT protocol. The methodology then checks the state space of the CPN corresponding to the PIT protocol with CPN-Attacks integrated to ensure correct execution. The formal verification result proves that the PIT protocol is robust against cyberattacks.

Building on the foundations of Project Cerberus (Microsoft, 2018), an open source initiative to create a hardware root of trust (*HRoT*) for server platforms, we have enhanced Cerberus's capabilities to develop a more robust security protocol. Although Cerberus effectively manages secure firmware attestations on devices, it does not inherently include lock-and-unlock mechanisms and user authentication features. Our implementation of the PIT protocol introduces these lock-and-unlock mechanisms linked to an authentication protocol, allowing for stricter control over the BIOS even before the boot-up phase. To facilitate the implementation process, we have designed a library called "*pit*", along with several APIs and comprehensive documentation. We have open-sourced this library, making it accessible for broader adoption and collaboration in the security community.

The rest of the paper is organized as follows. Section 2 explores the PIT protocol specifications, locking and unlocking mechanism, and its correctness property. Section 3 illustrates how the PIT protocol is modeled as a CPN and analyzes its correctness. Section 4 presents threat model generation using STRIDE and conversion to CPN attacks. Section 5 integrates CPN attacks into the CPN PIT model and uses reachability analysis to analyze the security of PIT protocol. Section 6 provides a detailed explanation of the implementation of the PIT protocol. Section 7 delivers the findings of our studies on current research. Section 8 summarizes our work and discusses future work.

2. Specification of PIT protocol

This section describes the PIT protocol architecture, the lock and unlock mechanisms, and the protocol's correctness property. It then uses a UML sequence diagram to show the interaction between the PIT protocol entities.

2.1. Overview of PIT protocol

PIT protocol incorporates five essential entities: *Device*, *HRoT*, *Server*, *Device Sender* (referred to as *Sender*) and *Device Recipient* (henceforth referred to as *Recipient*). *Device* (with a BIOS) is the protected entity that is safeguarded in transit by the PIT protocol.

HRoT is a tamperproof microcontroller embedded in the *Device* that locks and unlocks the *Device*'s BIOS. The PIT protocol is implemented in the *HRoT*. *Sender* is the entity that sends the *device*, such as a manufacturer or company. *Recipient* is the entity that receives the *device*. *Server* facilitates secure communication between *HRoT* and *Recipient*, ensuring that only authorized users can access the *Device*. Fig. 2 illustrates the relation between the PIT protocol entities. *Device* and *HRoT* are co-located in the physical product that is shipped by the *Sender*. *HRoT* and *Recipient* communicate with the *Server* over the Internet. PIT protocol is divided into the locking and unlocking phases.

Locking Phase: In this phase, the *HRoT* operates, but the BIOS is not operational and the remaining part of *Device* is not functional.

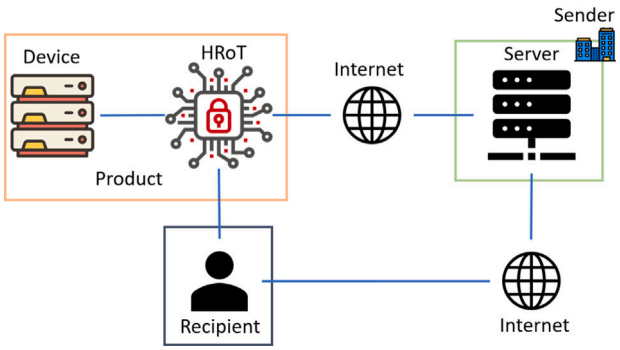

Fig. 2. Overview of PIT Protocol.

The locking phase is initiated by the *Sender* when *Device* is ordered and shipped to the *Recipient*. *Device* is locked by locking the BIOS; therefore, it is not responsive to typical user input. *Sender* sends the *Device* information (product ID, model, year and serial number) to *Server* for registration. *Server* creates and stores a unique Device Registration ID ($RegID$) locally in a database system. *Server* then associates this $RegID$ with an authorized and registered *Recipient*. *Server* sends the $RegID$ to *HRoT* who uses it to generate a pair of public/private keys using the Elliptic-Curve Diffie–Hellman (ECDH) key agreement protocol (Haakegaard and Lang, 2015); private key is d_A and the public key is Q_A, where $Q_A = d_A \times G$ and G is the base point of the chosen elliptic curve. *HRoT* sends the public key Q_A to *Server*. *Server* uses the *HRoT* public key Q_A to create a secret key by generating a random sequence r, computing a public key $R = r \times G$, and producing an AES secret key $S = r \times Q_A$. *Server* transmits its public key R to *HRoT*, which computes the same AES secret key S by performing the computation, $S = d_A \times R$. Note that $d_A \times R = r \times Q_A$. The secret key S is used for AES-GCM encryption and decryption to secure online interaction during the unlocking phase.

Unlocking Phase: The unlocking phase commences when the *Recipient* initiates an unlock request to the *Server*. The *Recipient* turns on and connects *HRoT* to *Server* before initiating the unlocking phase. *Server* looks up the registered *Recipient*, generates an encrypted registration ID, $RegID_s$, and sends it to the *HRoT*. *HRoT* decrypts the $RegID_s$ and validates it against the stored $RegID$. *HRoT* then creates a One-Time Password (*OTP*) for the *Recipient* and encrypts it using AES-GCM (Dworkin, 2007) with the secret key S. *HRoT* sends the encrypted *OTP*, denoted OTP_s, to *Server* and waits for *Recipient* to provide a valid OTP_s. *Server* receives OTP_s and sends it to the *Recipient* via an out-of-band channel, such as a secure email or mobile phone, ensuring secure communication. *Recipient* inputs the OTP_s to *Device*, and *Device* passes it to *HRoT* for validation. *HRoT* decrypts OTP_s as an *OTP* and validates *OTP*. *HRoT* sends an unlocking message to *Device* if *OTP* is valid; otherwise, it sends an invalid message to *Device*.

Correctness of the PIT Protocol: As the *Device* is shipped from *Sender* to the *Recipient*, there is a potential for tampering. The protocol aims to prevent *Device* tampering during transit. This is achieved by locking the *Device* via BIOS during transit and allowing it to be unlocked only by an authorized user after mutual authentication.

Property. *Recipient unlocks the correct Device if and only if the PIT protocol successfully authenticates the Recipient. This property is enumerated using four use cases:*

1. *If Recipient is malicious and the Device is incorrect, the PIT protocol prevents unlocking the Device.*
2. *If Recipient is legitimate and the Device is incorrect, the PIT protocol prevents unlocking the Device.*
3. *If Recipient is malicious and the Device is correct, the PIT protocol prevents unlocking the Device.*
4. *If Recipient is legitimate and the Device is correct, the PIT protocol unlocks the Device.*

2.2. UML sequence diagrams of PIT protocol

The UML Sequence Diagrams describe the interactions between the PIT protocol's entities using message communications. We express the UML Sequence Diagram as an 8-tuple, $SD = \langle \mathcal{E}, \mathcal{L}, \mathcal{F}, \mathcal{R}, \mathcal{M}, \mathcal{D}, \mathcal{T}, \mathcal{P} \rangle$, where

- $\mathcal{E}$ is a set of entities; $\mathcal{E} = \{e_1, e_2, \dots, e_n\}$.
- $\mathcal{L}$ is a set of lifelines.
- $\mathcal{F}$ is a set of combined fragments.
- $\mathcal{R}$ is a function identifying a fragment type; $\mathcal{R} : \mathcal{F} \rightarrow (alt, loop, opt, par, break, \dots)$.
- $\mathcal{M}$ is a set of messages; $\mathcal{M} = \{m_1, m_2, \dots, m_n\}$.
- $\mathcal{D}$ is a function that determines the order of a message $m \in \mathcal{M}$ on the lifeline $\mathcal{L}$; $\mathcal{D} : (\mathcal{L}, \mathcal{M}) \rightarrow \mathbb{N}$.
- $\mathcal{T}$ is a mapping function that is used to determine a message type; $\mathcal{T} : \mathcal{M} \rightarrow (Syn, Asyn)$.
- $\mathcal{P}$ is a set of parameters; $\mathcal{P} = \{p_1, p_2, \dots, p_i\}$.

A message comprises a set of parameter-value pairs as $m_i(p_1 = v_1, p_2 = v_2, \dots, p_i = v_i)$. The message types include synchronous and asynchronous messages. The set of PIT protocol entities are $\mathcal{E} = \{HRoT, Device, Server, Sender, Recipient\}$. The UML sequence diagram of the locking phase, shown in Fig. 3, includes *Sender*, *Server*, and *HRoT* entities representing three lifelines as $\mathcal{L}_{lock} = \{HRoT, Sender, Server\}$.

The locking phase does not have combined fragments; hence, $\mathcal{F}_{lock} = \{\emptyset\}$. The messages are $\mathcal{M}_{lock} = \{ProductRegistration(Info), Send(RegID), SendPublicKey(qA), SendPublicKey(qB)\}$, and the parameters are $\mathcal{P}_{lock} = \{Info, RegID, qA, qB\}$. The unlocking phase, shown in Fig. 4, incorporates *Recipient*, *Device*, *Server*, and *HRoT* entities represented as four lifelines as $\mathcal{L}_{unlock} = \{HRoT, Device, Recipient, Server\}$. The unlocking phase has two alt-fragments $\mathcal{F}_{unlock} = \{Validate(RegID), Validate(OTP)\}$. The set of messages is $\mathcal{M}_{unlock} = \{UnlockRequest(credential), Send(RegIDs), Valid(msg), Invalid(msg), Send(OTPs), Input(OTPs), Pass(OTPs), Unlock(), Request(accepted), Request(rejected)\}$, and the parameters $\mathcal{P}_{unlock} = \{credential, RegIDs, OTPs, msg, accepted, rejected\}$.

The sets of entities, lifelines, messages, and parameters construct two tuples, each describing a UML sequence diagram of the PIT protocol phase. These tuples are defined as: $SD_{lock} = \langle \mathcal{E}_{lock}, \mathcal{L}_{lock}, \mathcal{F}_{lock}, \mathcal{R}, \mathcal{M}_{lock}, \mathcal{D}, \mathcal{T}, \mathcal{P}_{lock} \rangle$ and $SD_{unlock} = \langle \mathcal{E}_{unlock}, \mathcal{L}_{unlock}, \mathcal{F}_{unlock}, \mathcal{R}, \mathcal{M}_{unlock}, \mathcal{D}, \mathcal{T}, \mathcal{P}_{unlock} \rangle$.

This formalization of the UML sequence diagrams is transformed into the CPN to automate the verification of the PIT protocol.

3. CPN modeling of PIT protocol and correctness analysis

We design a methodology that systematically follows a logical progression of 7 interdependent steps, each building upon the outcomes of the previous one. As illustrated in Fig. 5, it starts with the PIT protocol specification as input and ends with a security analysis of the PIT protocol.

Step 1: PIT Protocol specifications, uses UML sequence diagrams to visualize the specifications and derive entities, messages, and parameters. These elements are the building blocks for transforming the PIT protocol into a CPN and Data Flow Diagram (DFD), marking the beginning of the methodology's process.

Step 2: CPN PIT Protocol, uses the output of Step 1 to generate the CPN for the PIT protocol. This CPN will be used to analyze the correctness of the PIT protocol.

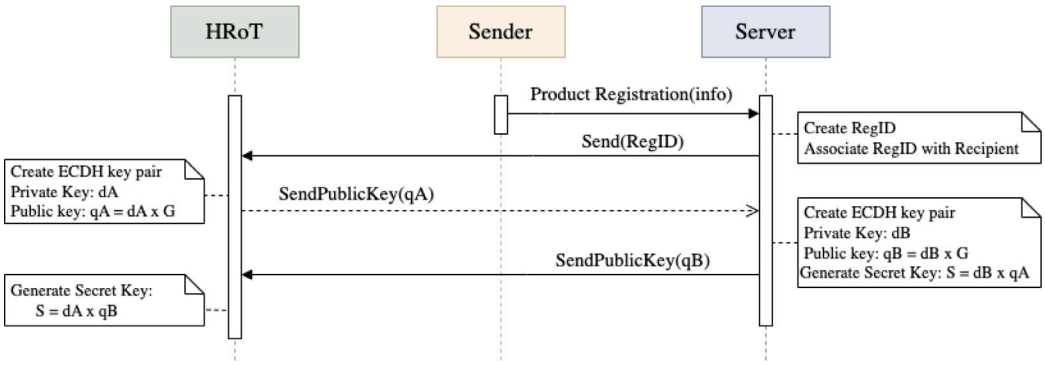

Fig. 3. Locking Phase of PIT Protocol.

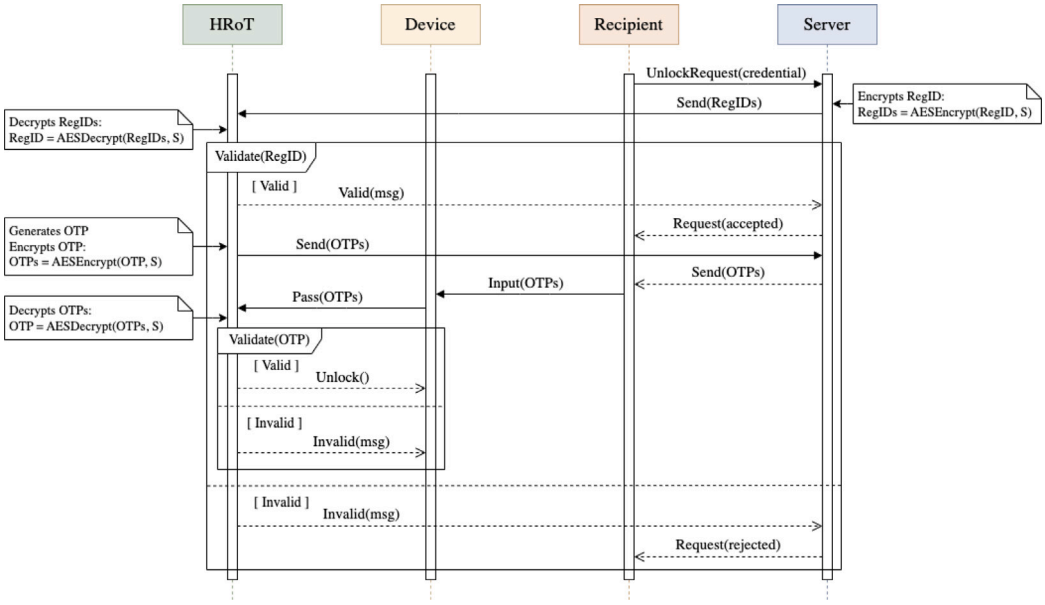

Fig. 4. Unlocking Phase of PIT Protocol.

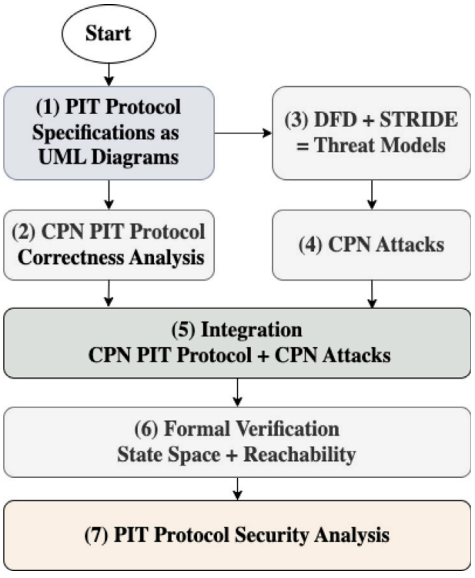

Fig. 5. Analysis Methodology.

Step 3: Step 3 takes the output of Step 1 and generates DFD, which the STRIDE framework uses to generate threat models.

Step 4: These threat models are then used in Step 4 to generate CPN attacks.

Step 5: he results of Steps 2 and 4 are required to perform Step 5 (Integration of the CPN PIT Protocol and CPN attacks), where CPN attacks are integrated with the CPN PIT protocol to present the effect of various cyberattack scenarios.

Step 6: In Step 6 the integration of the CPN PIT protocol with the CPN attacks, is formally verified. The state space is generated for each cyberattack scenario and the state transitions are verified.

Step 7: Based on formal verification, Step 7 analyzes the security of the PIT protocol and highlights the findings.

The remainder of this section addresses Step 2, deriving CPN for the PIT protocol and analyzing its correctness. The process of generating threats and transforming them into CPN attacks (Steps 3 and 4) is detailed in Section 4, while the security analysis (Steps 5, 6, and 7) is presented in Section 5.

3.1. Derive CPN PIT protocol from UML sequence diagrams

A CPN is a bipartite directed graph where the nodes correspond to places P, transitions T, and arcs A are directed edges from a place to a transition or from a transition to a place. The input place of the transition is where a directed arc exists between the place and the transition. CPN operates on multisets of typed objects called tokens. The places are assigned tokens at initialization. Transitions consume tokens from their input places, perform some action, and output tokens on their output places. The distribution of tokens over the places of the CPN defines the states, referred to as markings. Transitioning from one marking to another represents the system's state transition (i.e., state space). The CPN is formally defined as:

Definition 1 (*Coloured Petri Nets (CPN)*)**.** CPN is defined as 9-tuple $CPN = \langle P, T, A, \Sigma, V, C, G, E, I \rangle$, where

- P is a finite set of places.
- T is a finite set of transitions.
- A is a set of directed arcs $A \subseteq (P \times T) \cup (T \times P)$.
- Σ is a finite set of color sets, corresponding to types.
- V is a finite set of type variables, and $Type[v] \in \Sigma$ for $\forall v \in V$.
- C is the color function, which is the mapping from place set to color set such that $C : P \rightarrow \Sigma$.
- G is the guard function, which is the mapping such that $G : T \rightarrow EXPR$, and $Type[G(t)] = Bool$.
- E is an arc expression function, which is the mapping such that $E : A \rightarrow EXPR$, and $Type[E(a)] = C(p)_{MS}$, where p is the place connected to an arc and MS represents the polymorphic set.
- I is the initial function of p, which is the mapping such that $I : P \rightarrow EXPR$, and $Type[I(p)] = C(p)_{MS}$.

Algorithm 1 converts each PIT UML Sequence Diagram tuple into a CPN tuple as:

$$SD = \langle \mathcal{E}, \mathcal{L}, \mathcal{F}, \mathcal{R}, \mathcal{M}, \mathcal{D}, \mathcal{T}, \mathcal{P} \rangle \xrightarrow{Algorithm\ 1} CPN = \langle P, T, A, \Sigma, V, C, G, E, I \rangle$$

It takes a sequence diagram tuple, SD, as input and generates the CPN tuple, CPN. It first iterates over SD parameters, converting them into colors (CPN datatypes) and creating CPN variables for each color (lines 1 to 6). The CPN model is then built by initializing the CPN place-transition pair as the starting point for each SD lifeline as a CPN branch (lines 7 to 12). For each SD lifeline, Algorithm 1, in lines 13 to 29, iterates over messages, each of which is converted into a CPN block (place, transition, place) and is connected with arcs. Each CPN block is connected to the previous CPN block. This process repeatedly covers all SD lifelines and then connects one CPN branch to another through CPN transitions. Algorithm 1, in lines 30 to 37, runs over each SD combined fragment, recursively converting it into a CPN sub-model. Each SD combined fragment type represents a CPN transition with guard expression. Then, the CPN sub-model is connected to the main CPN through two special places (In-Port and Out-Port) to construct a hierarchical CPN model. Through lines 38 to 43, the algorithm creates a CPN transition for each branch to connect them into one final place.

Figs. 6, 7, and 8 show the result of Algorithm 1 converting the UML sequence diagrams of the locking phase into three CPNs, *Sender*, *Server*, and *HRoT* respectively. The token of the product, colored green at the top of Fig. 6 for CPN *Sender* expresses the information of the product that must be locked by *HRoT* and is shipped to *Recipient*. We instantiate this token of the *Sender* CPN as $\{deviceID = \}\}HP-12345-2023\varepsilon,\ DeviceName = \}\}MB-DX-34\varepsilon,\ DeviceVersion = \}\}v1.231.2023\varepsilon, SerialNo = \}\}SN-9876-2023-018\varepsilon, Company = \}\}HP\varepsilon, ShipDate = \}\}02-12-2024\varepsilon,\ Status = PROCESSED\}$. The *Server* uses this product token to create a registration ID ($RegID$). Another token, representing the *Recipient* credential (user name and password), is placed in the *Server* CPN and associated with $RegID$.

Algorithm 1: UML Sequence Diagram to CPN

Input : $SD = \langle \mathcal{E}, \mathcal{L}, \mathcal{F}, \mathcal{R}, \mathcal{M}, \mathcal{D}, \mathcal{T}, \mathcal{P} \rangle$
Output: $CPN = \langle P, T, A, \Sigma, V, C, G, E, I \rangle$

/* convert SD parameters to CPN colors and variables */
1 **foreach** $pr \in SD.\mathcal{P}$ **do**
2 $\Sigma \leftarrow add(color : Type(pr))$;
3 **if** (pr *has initial value*) **then**
4 $V \leftarrow add(v_i : Type(pr))$;
5 **end**
6 **end**
/* start building CPN model, Initialize place-transition pair */
/* and, branch by arcs with number of liveliness */
7 $P \leftarrow addPlace(p_{init})$;
8 $T \leftarrow addTransition(t_{init})$;
9 $A \leftarrow addArc(a_{init})$;
10 $connect(p_{init}, a_{init}, t_{init})$;
11 $A \leftarrow addAllArcs(a_i^{out}, size(SD.L))$;
12 $connectAll(t_{init}, a_i^{out})$;
/* for each liveline, iterate over messages (sync and return) */
/* convert each message to CPN block (place, transition, place) */
/* and then connect them with arcs */
13 **foreach** $l_i \in SD.\mathcal{L}$ **do**
14 **foreach** $m_j \in SD.\mathcal{M}$ **do**
15 $T \leftarrow addTransition(t_j)$;
16 $A \leftarrow addArc(a_j^{out})$;
17 $P \leftarrow AddPlace(p_j^{out})$;
 /* is there a previous block? */
 /* No, create a new CPN block */
18 **if** T *is Empty* **then**
19 $P \leftarrow AddPlace(p_j^{in})$;
20 $A \leftarrow addArc(a_j^{in})$;
21 $connect(p_j^{in}, a_j^{in}, t_j, a_j^{out}, p_j^{out})$;
22 **end**
 /* otherwise, connect with previous CPN block */
23 **else**
24 $A \leftarrow addArc(a_j^{in})$;
25 $connect(p_{j-1}^{out}, a_j^{in}, t_j, a_j^{out}, p_j^{out})$;
26 **end**
27 **end**
28 $CPN \leftarrow CPN \cup \{P, T, A, \Sigma, V, C, G, E, I\}$;
29 **end**
/* recursively, for each combined fragment, create sub CPN */
/* and then connect it to the supper CPN model */
30 **foreach** $f_k \in SD.\mathcal{F}$ **do**
31 $T \leftarrow addTransition(t_k^{sub})$;
32 $G : addGuardExpession(t_k^{sub},\ R : Type(f_k))$;
33 $Recursive(f_k, CPN_{sub})$;
34 $createInPort(P \leftarrow AddPlace(p_k^{inPort}))$;
35 $createOutPort(P \leftarrow AddPlace(p_k^{outPort}))$;
36 $connect(CPN, CPN_{sub}, p_k^{inPort}, t_k^{sub}, p_k^{outPort})$;
37 **end**
/* create final transitions to connect to one final place */
38 **foreach** $p_i \in P$ **do**
39 $A \leftarrow addArc(a_i^{final})$;
40 $T \leftarrow addTransition(t_i^{final})$;
41 **end**
42 $P \leftarrow AddPlace(p^{final})$;
43 $connectAll(a^{final}, t^{final}, p^{final})$;
44 **return** CPN

This token is instantiated as {userName="test@gmail.com", Password ="Test12345"}. The product token will be sent from the *Sender* CPN to the *Server* to generate $RegID$ and associate it with the *Recipient* credential token. The *Server* CPN passes the $RegID$ to the *HRoT* CPN to generate public and private keys. The *Server* and *HRoT* CPNs then exchange public keys to secure the channel for the unlocking phase. Once the public keys are exchanged, the *HRoT* CPN locks the *Sender*. Figs. 9, 10, and 11 show three CPNs of the unlocking phase. Fig. 9 represents the unlocking phase of the *Recipient* and the *Device* as one CPN. Figs. 10 and 11 represent the unlocking phase of the *HRoT* and the *Server* respectively.

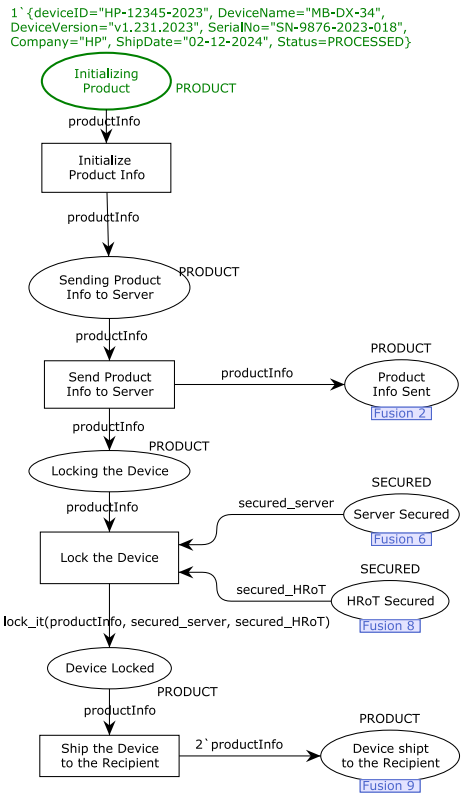

Fig. 6. Locking Phase - CPN for Sender.

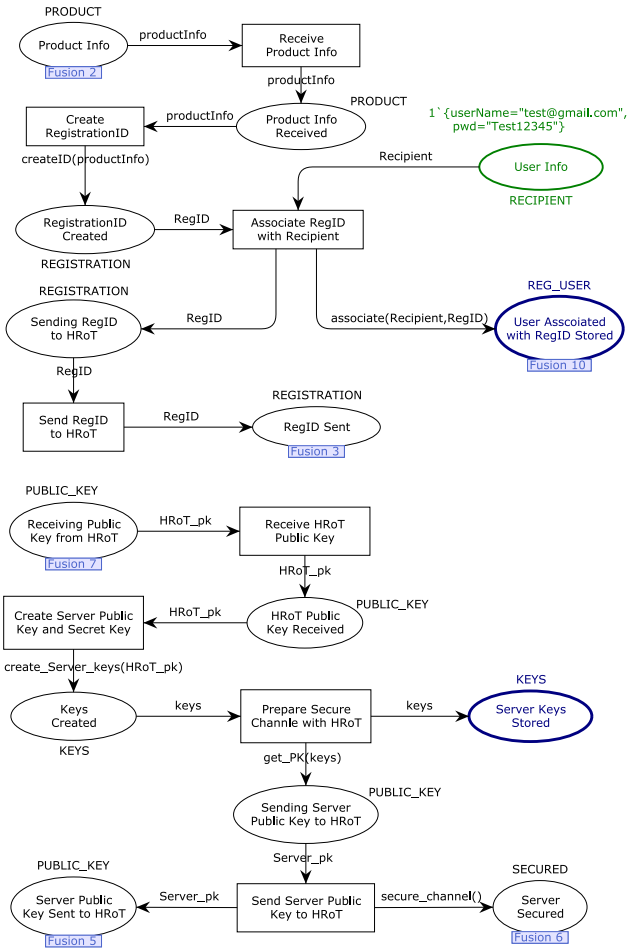

Fig. 7. Locking Phase - CPN for Server.

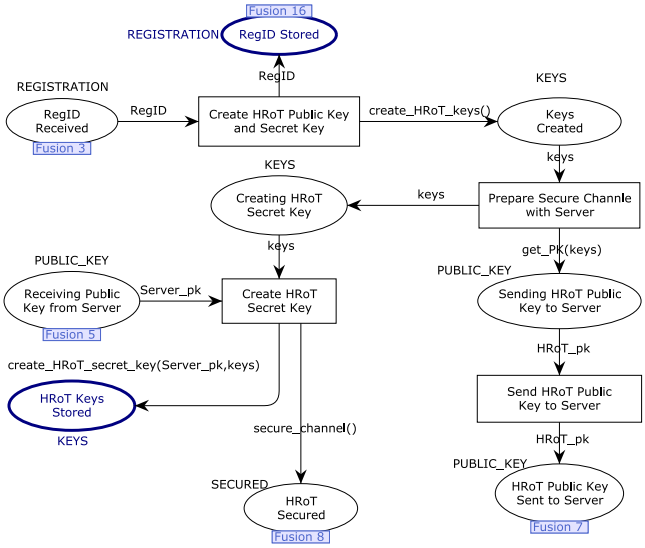

Fig. 8. Locking Phase - CPN for HRoT.

3.2. Correctness analysis of the PIT protocol

The correctness of the PIT protocol is verified through the CPN model using CPN Tools (Ratzer et al., 2003). We use the CPN-ML programming language (Jensen and Kristensen, 2009), which is associated with CPN Tools, to verify dead markings and dead transitions for the CPN of the PIT protocol. We also utilize built-in functions provided by the CPN Tools, including *Reachable*(x, y), *AllReachable()*, *NodesInPath*(x, y), *DeadMarking*(x), *ListDeadMarkings()*, and *ListDeadTransitions()*. These functions are essential for verification and reachability analysis. For instance, the *ListDeadMarkings()* function enumerates all possible dead markings (termination states). The *NodesInPath*(x, y) function retrieves a path (an execution sequence of system states) from x as an initial state and y as one of these dead markings. These paths are instrumental in determining where an access request is executed, where a successful response is received, and where it is denied. They also help identify scenarios where an access request is executed, but no response is received.

The CPN Tools also simulate the state transitions and interactions of the PIT protocol between its entities (*Device, HRoT, Server, Sender and Recipient*) during the lock and unlocking phases. The verification process is done through simulation and formal analysis:

Simulation: The CPN model of the PIT protocol is simulated to capture the state transitions during the protocol execution. This allows us to observe how tokens are passed between various entities and ensure that the PIT protocol progresses as expected under different scenarios, including encryption and decryption operations and authentication conditions.

Formal Analysis: Two aspects of the formal analysis were applied to the CPN model of the PIT protocol:

1. **State Space Analysis:** We examine the state space generated from the CPN model of the PIT protocol to identify cyclic, deadlocked and unreachable states. We also analyze dead markings and transitions to ensure that the PIT protocol terminates correctly in all cases and does not reach invalid states, where the CPN model finishes its process before the PIT protocol process is fully completed. For instance, in the CPN model for the

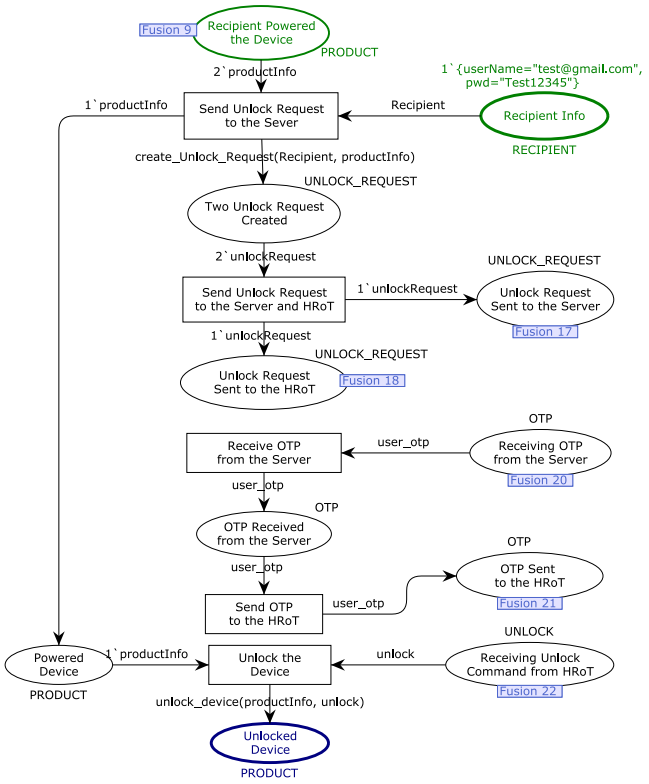

Fig. 9. Unlocking Phase - CPN for Recipient & Device.

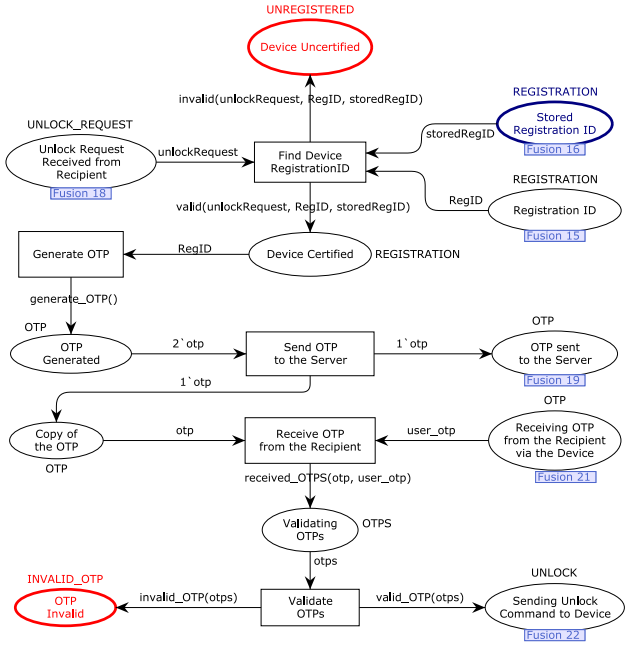

Fig. 10. Unlocking Phase - CPN for HRoT.

unlocking phase, it receives an unlock request from *HRoT* and successfully verifies the *Device*, but it terminates before sending the OTP to validate the *Recipient*. This premature termination of the CPN model would be considered an invalid state.

2. **Reachability Analysis:** We conduct a thorough reachability analysis to explore all possible states of the PIT protocol, ensuring that every valid state is reachable and that no undesired states are encountered during protocol execution. We utilize the

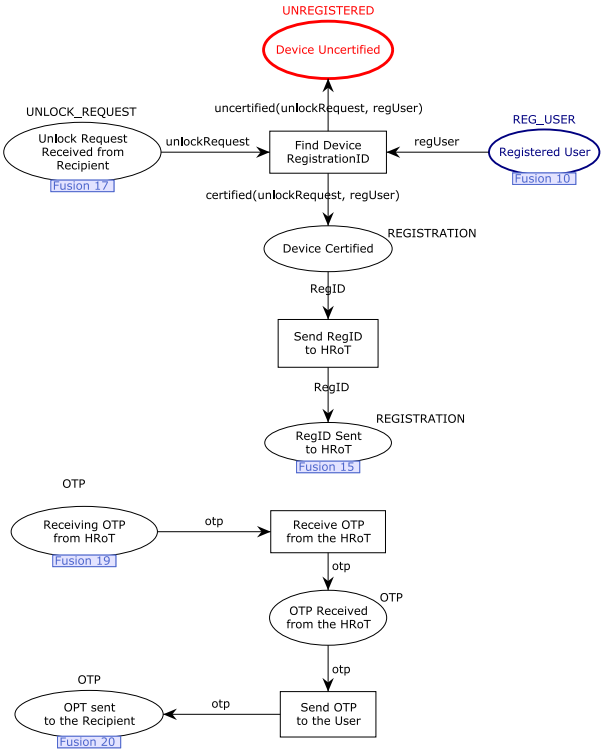

Fig. 11. Unlocking Phase - CPN for Server.

Reachable(x, y) function, where x represents an initial state and y represents a destination state, which determines if it is an undesired state (e.g., illegal for user to access) is reachable. The reachability analysis helps us refine the CPN model and ensure that the protocol correctly prevents illegal users from accessing the device.

In this phase, we conducted the verification process on the CPN model of the PIT protocol, assuming that no attacks have occurred. Our priority in this phase is to confirm that the CPN model functions correctly. The green circles are CPN places where the initial values (tokens) of the PIT protocol process are assigned. The red circles are CPN places where the *Recipient* fails to provide the correct credentials or *OTP* (CPN terminations). This verification confirms that the PIT protocol behaves correctly; the rightful *Recipient* unlocked the correct *Device*, ensuring the correctness property of the PIT protocol.

4. Threat modeling

The PIT protocol is built on several assumptions. We assume the *HRoT* processor is tamper-proof and fully trusted, ensuring it has not been compromised during manufacturing or shipment. This assumption signifies that the *HRoT* is "clean" when the PIT protocol is installed, implying it is free from malicious firmware and hardware modifications. The integrity of the *HRoT* is crucial, as it serves as the foundational security aspect for PIT protocol, ensuring it is immune to tampering during deployment and transit. We also assume that the company (*Sender*) is a trusted entity with no risk of insider threats originating from within the organization. Furthermore, we consider the *Server* to be a trusted zone. In the event of attacks that could lead to *Server* unavailability, such scenarios will not be included in the security analysis. We now elaborate on how to derive DFD from the UML sequence diagrams and use the Microsoft Threat Modeling Tool (MTMT) (Shostack, 2014) to generate threat models.

4.1. Convert UML sequence diagram to DFD and apply STRIDE

DFD includes four elements: process, data flow, data store, and external entities or actors. The entities defined in the PIT protocol specifications are mapped to DFD as processes. The UML sequence diagrams express the PIT protocol entities' interaction flow. They are used to generate DFD data flows. The data store described in the protocol specifications is a DFD data store. An entity that interacts with the protocol specifications but is not part of it is represented as a DFD actor.

Definition 2 (*Data Flow Diagram (DFD)*)**.** We represent the DFD as a 4-tuple, $DFD = \langle \mathcal{P}^D, \mathcal{A}^D, \mathcal{S}^D, \mathcal{F}^D \rangle$, where:

- $\mathcal{P}^D$ is a set of processes in DFD
- $\mathcal{A}^D$ is a set of actors or entities in DFD
- $\mathcal{S}^D$ is a set of data stores in DFD
- $\mathcal{F}^D$ is a set of data flows in DFD

Algorithm 2 converts the UML Sequence Diagram (SD) tuples into a DFD tuple as:

$$SD = \langle \mathcal{E}, \mathcal{L}, \mathcal{F}, \mathcal{R}, \mathcal{M}, \mathcal{D}, \mathcal{T}, \mathcal{P} \rangle \xrightarrow{Algorithm\ 2} DFD = \langle \mathcal{P}^D, \mathcal{A}^D, \mathcal{S}^D, \mathcal{F}^D \rangle$$

It takes a UML sequence diagram tuple, SD, as input and generates a DFD tuple, DFD. Algorithm 2 first initializes an empty DFD with sets for processes, entities, data stores, and data flows: $DFD = \{Processes : \{\}, Actors : \{\}, DataStores : \{\}, DataFlows : \{\}\}$. For each entity in SD, it creates a corresponding entity in DFD and adds it to the DFD's actors set as $Actors : \{HRoT, Device, Server, Recipient\}$. Similarly, for each lifeline in the UML SD, it creates a corresponding process in the DFD with the same name and adds it to the DFD's processes set as $Processes : \{Lock, Unlock\}$. It also creates a data store element if a process involves storing data as $DataStores : \{dataStore\}$. For each message in SD, Algorithm 2 identifies the sender and receiver by iterating over the DFD elements and creating a data flow with the message's name, sender, receiver, order, type, and parameters. It adds this data flow to the DFD's data flow set. If a message involves storing data, it creates a corresponding data store in DFD and adds it to DFD's data store set. For each combined fragment in SD that is not empty, Algorithm 2 creates a corresponding process in DFD with the fragment's name and adds it to the DFD's processes set. After processing all SD elements, Algorithm 2 returns the constructed DFD.

The DFD of the PIT protocol shown in Fig. 12 illustrates the DFD generated from the UML sequence diagrams of the locking and unlocking phases. We manually identify any external entity, create a data flow, link it to the corresponding process, and add it to the DFD's data flow set. For instance, the *Database* entity shown in Fig. 12 is not part of the PIT protocol; however, it is an external entity described in the specification as part of the *Server* and required for product registration. Such external entities may reveal potential threats that we want to consider. Thus, it is manually added to the DFD's data flow set.

MTMT (Shostack, 2014) uses DFD elements to generate threat models and categorizes them as Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege. The MTMT also reports the threat priority as low, medium, and high. The high-priority threats are selected for cyberattack demonstration.

Fig. 12 illustrates 16 flows that describe the PIT protocol's DFD locking and unlocking phases. The locking phase starts from flows number 1 to 4. The *Sender* sends the product information to the *Server* to register the Device (flow number 1). The *Server* creates $RegID$ and sends it to the *HRoT* (flow number 2). Then, *HRoT* and *Server* exchange public keys (flow numbers 3 and 4). This ends the flow of the locking phase. The locking flows are not considered threats. These actions are merely system activation steps and occur in a secure environment before the Device is shipped.

Algorithm 2: UML Sequence Diagram to DFD

Input : $SD = \langle \mathcal{E}, \mathcal{L}, \mathcal{F}, \mathcal{R}, \mathcal{M}, \mathcal{D}, \mathcal{T}, \mathcal{P} \rangle$ (UML Sequence Diagram)
Output: $DFD = \langle \mathcal{P}^D, \mathcal{A}^D, \mathcal{S}^D, \mathcal{F}^D \rangle$ (Data Flow Diagram)

```
/* identify Actors and External entities */
foreach e ∈ SD.E do
    actor ← {name: e};
    A^D ← A^D ∪ {actor};
    DFD.elements ← DFD.elements ∪ {actor};
end
/* identify lifelines as Processes */
foreach l ∈ SD.L do
    process ← {name: l};
    P^D ← P^D ∪ {process};
    DFD.elements ← DFD.elements ∪ {process};
    /* identify messages as Data Flows, Data Stores, and
       Handle Return messages */
    foreach m ∈ SD.M sorted by SD.D(m) do
        /* find sender and receiver */
        sender ← null;
        receiver ← null;
        foreach element ∈ DFD.elements do
            if element.name = m.sender then
                sender ← element;
            end
            if element.name = m.receiver then
                receiver ← element;
            end
        end
        /* handle message based on type */
        if SD.T(m) = Syn then
            /* create Data-flow for Syn type */
            dataFlow
              ← {name : m.name, sender : sender, receiver : receiver, order :
              SD.D(m), type : SD.T(m), parameters : m.parameters};
        end
        else if SD.T(m) = return then
            /* swap sender and receiver for return type */
            dataFlow
              ← {name : m.name, sender : receiver, receiver : sender, order :
              SD.D(m), type : SD.T(m), parameters : m.parameters};
        end
        /* add Data-flow to the DFD */
        F^D ← F^D ∪ {dataFlow};
        DFD.elements ← DFD.elements ∪ {dataFlow};
        /* create Data-store if involved */
        if m.involvesDataStore then
            dataStore ← {name : m.name};
            S^D ← S^D ∪ {dataStore};
            DFD.elements ← DFD.elements ∪ {dataStore};
        end
    end
end
/* handle combined fragments */
foreach f ∈ SD.F do
    if SD.R(f) ≠ empty then
        process ← {name : f};
        P^D ← P^D ∪ {process};
        DFD.elements ← DFD.elements ∪ {process};
    end
end
return DFD;
```

We consider threats associated with the unlocking phase when the Device is in transit and arrives at the Recipient. The unlocking phase is covered by flow numbers 5 to 16. The *Recipient* raises an unlock request by turning on the *HRoT* and connecting to the *Server* using credentials (flow numbers 5 and 6). The *Server* checks the Recipient's credentials and fetches the $RegID$ associated with the Recipient from the database (flow numbers 7 and 8). The *HRoT* sends encrypted $RegID$ ($RegIDs$) to the *Server* for device validation (flow numbers 9 and 10). For the Recipient validation (flow numbers 11 to 12), the *HRoT* sends *OTPs* to the *Server*, which sends them to the Recipient's personal contact (e.g., a phone text message). The Recipient inputs the *OTPs* into the *HRoT*, and the *HRoT* validates it (flow numbers 13 and 14). If the two validations are successfully passed, the *HRoT* unlocks the Device through the I2C channel (flow numbers 15 and 16). The I2C channel between the Device and *HRoT* is physically secure, and no physical tampering is possible since this communication is internal to the Device; physical tampering of the channel is considered outside the threat model. The MTMT threats that are high priority are summarized below.

Threats on HRoT:

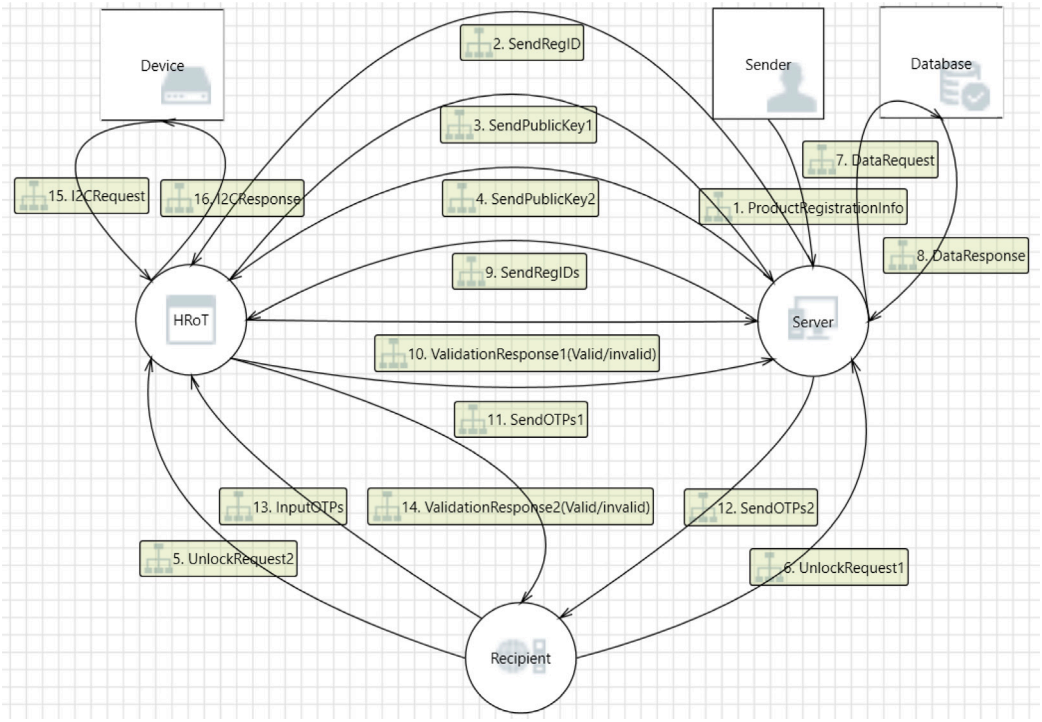

Fig. 12. DFD of the PIT Protocol.

- Spoofing: The [$RegIDs$] can be spoofed, disrupting the *Device*'s unlocking process (Fig. 12, flow number 9).
- Tampering: An attacker can tamper data ([$RegIDs$]) to launch a Man-in-the-Middle (MitM) attack and disrupt *Device* (flow number 9).

Threats on Server:

- Spoofing: The [$RegIDs$] validation response can be spoofed to disrupt the *Device*'s unlocking process (flow number 10).
- Elevation of Privilege: By stealing *Recipient* credentials, the attacker can gain unauthorized privileges on the *Server* (flow number 6).
- DoS: An attacker can gain unauthorized access to *Server*, launch a denial of service attack, and lead to *Server* unavailability (flow number 8).

Threats on Recipient:

- Spoofing: The [*OTPs*] validation response can be spoofed by an attacker, disrupting the *Device*'s unlocking process (flow number 12).
- Tampering: The spoofed data can be tampered with to launch MitM attacks by impersonating the *Server* (flow numbers 6 and 12).
- Repudiation: By injecting malicious/false [*OTPs*], the attacker tries to gain unauthorized access to *HRoT* (flow number 12).

Threats on HRoT–Server and Recipient–Server Communication Channels:

- Spoofing: The attacker can capture packets containing [$RegIDs$] and [*OTPs*] and launch MitM attacks (flow numbers 9, 11, and 12).
- Tampering: Tampering data in communication channels can disrupt the execution flow (flow numbers 8–12).
- Repudiation: By injecting malicious/false data into the captured packets, the attacker can gain unauthorized access, control, and manipulate *Server* and *HRoT* (flow numbers 9–12).

These threats are converted into CPN attacks to demonstrate various cyberattack scenarios.

4.2. Attacker capabilities

To launch attacks by exploiting these threats, we assume that the attacker possesses specific capabilities tailored to each type of attack discussed below.

For the spoofing attack on *HRoT*, the attacker needs network access to intercept or inject spoofed [$RegIDs$] into the communication stream. An attacker can compromise the authenticity of [$RegIDs$] by manipulating or forging *Recipient* credentials and potentially bypassing integrity checks or cryptographic measures. For a tampering attack, the attacker can intercept communication between *HRoT* and the trusted *Server*, acting as an intermediary. The attacker can alter [$RegIDs$] in transit without detection, requiring control over the communication channel—possibly through network hijacking or compromising routers. The attacker may bypass encryption mechanisms by decrypting traffic or forging cryptographic keys.

For the elevation of privilege type attacks on the *Server*, the attacker can potentially obtain valid credentials through phishing, keylogging, or weak authentication exploits. Once inside, they can use privilege escalation techniques to exploit system vulnerabilities or misconfigurations. A good understanding of the *Server*'s architecture is essential for effective exploitation. The attacker can overwhelm *Server* with a flood of requests, often using botnets – a network of compromised devices – to disrupt service thus triggering a successful DoS attack.

A malicious *Recipient* can intercept a legitimate *OTP* through Man-in-the-Middle (MitM) attacks or generate false *OTPs* by brute-force or exploiting weak algorithms to launch a repudiation attack. A deep understanding of *HRoT*'s *OTP* verification is required to bypass its authentication mechanism.

By compromising communication channels, the attacker can access the channels between the *HRoT*, *Server*, and *Recipient* through MitM attacks, network sniffing, or compromise routers. If the communication is encrypted or secured with integrity checks or weak encryption schema, the attacker can bypass these protections by decrypting messages or

forging cryptographic signatures. A solid understanding of communication protocols can potentially lead to tamper with data without detection.

Under these various adversarial conditions, we want to investigate the PIT protocol resiliency; thus, next, we convert threats generated by STRIDE into CPN attacks.

4.3. Convert threats into CPN attacks

An attack is an action taken by an adversary that changes the state of an entity, rendering it unable to execute a process or provide a service. Each threat generated by STRIDE is designed as a CPN attack.

Definition 3 (*Threat Model*)**.** We express the threat model as an 4-tuple, $\mathcal{TR} = \langle \mathcal{E}, \mathcal{H}, C, \mathcal{W} \rangle, where$

- $\mathcal{E}$ is a set of entities
- $\mathcal{H}$ is a set of threats; $\mathcal{H} = \{h_1, h_2, \ldots, h_n\}$
- C is a set of variable-value pairs; $C = \{c_1(para_1 = v_1), c_2(para_2 = v_2), \ldots, c_n(para_n = v_n)\}$
- $\mathcal{W}$ assigns a threat to an entity; $\mathcal{W} : \mathcal{W} \subseteq (H \times E)$

A CPN attack is specified as the CPN place (precondition), transition (action), and place (postcondition). The preconditions and postconditions are conditions that must be true for CPN attacks to be integrated into the CPN PIT protocol. Algorithm 3 converts the threat model tuples into a CPN attack tuple as:

$$\mathcal{TR} = \langle \mathcal{E}, \mathcal{H}, C, \mathcal{W} \rangle \xrightarrow{Algorithm\ 3} CPN\_Attack = \langle P, T, A, \Sigma, V, C, G, E, I \rangle$$

Algorithm 3: Threat Models to CPN Attacks

```
Input  : TR = ⟨H, E, C, W⟩
Output: CPN_Attack = ⟨P, T, A, Σ, V, C, G, E, I⟩
/* for each entity, iterate over variables     */
/* and, iterate over threats reported by STRIDE
   */
foreach e_i ∈ TR.E do
    foreach h_j ∈ TR.H do
        /* for each threat, create CPN attack as:
           */
        /* (pre-place, attack-transition,
           post-place) and connect them with arcs
           */
        foreach c_k ∈ TR.C do
            P ← AddPlace(p^pre_[k,j]);
            A ← addArc(a^in_[k,j]);
            T ← addTransition(t^attack_[k,j]);
            A ← addArc(a^out_[k,j]);
            P ← AddPlace(p^post_[k,j]);
            connect(p^pre_[k,j], a^in_[k,j], t^attack_[k,j], a^out_[k,j], p^post_[k,j]);
            CPN_Attack ←
              CPN_Attack ∪ {P, T, A, Σ, V, C, G, E, I};
        end
    end
end
return CPN_Attacks
```

It takes a threat model tuple, TR, as input and generates a CPN attack tuple, CPN_Attack. Algorithm 3 comprises three nested iterations. It iterates over TR entities and, for each, iterates over threats reported by STRIDE. Each threat generates a CPN attack as (pre-place, attack-transition, post-place). For illustration, we select 6 CPN attacks, shown in Fig. 13, as Spoofing, Tampering, Elevation of Privilege, Repudiation, and Denial of Service (DoS). These CPN attacks are integrated into the CPNs of the PIT protocol to demonstrate various cyberattack scenarios.

Each CPN attack has specific preconditions and postconditions that must be satisfied to be integrated into the PIT protocol's CPN model. These conditions include the data types of input and output places where the CPN attack is to be incorporated. In addition, the variables declared with these data types must match the token values to execute the transition that represents the attack. For the spoofing attack on *Recipient*'s credentials (Fig. 13, the top sub-figure), the USER data type, which contains two variables, user name, and password, is a key precondition. The postconditions of this attack also have to be USER data type, including user name and password variables. These variables are passed to the output place when the spoofing attack transition is fired.

5. Security analysis

This section provides a security analysis of the PIT protocol, evaluating its resilience against cyberattacks. CPN attacks derived from the STRIDE framework are integrated into the CPN model to assess the protocol's ability to maintain mutual authentication and device integrity during transit against various attack scenarios.

5.1. Integrate CPN PIT protocol and CPN attacks

The integration of CPN attacks with the CPN PIT protocol presents various cyberattack scenarios. A cyberattack scenario refers to one or more CPN attacks attached to the CPN PIT protocol. The possible number of cyberattack scenarios is counted as follows:

$$|Attack\_Scenarios| = \sum_{i=1}^{n}((|E_i|)! + 1) + \prod_{i=1}^{n}((|E_i|)! + 1)$$

where n is the number of entities and $|E_i|$ is the number of threats in entity E_i. For instance, MTMT reported 3 threats in *HRoT*, 4 threats on *Server*, 3 threats on *Recipient*, and 3 threats on *HRoT–Server* and *Recipient–Server* Communication Channels. These threats are converted into 13 CPN attacks, and the possible number of combination cyberattack scenarios is $((3! + 1) + (4! + 1) + (3! + 1) + (3! + 1)) + ((3! + 1) \times (4! + 1) \times (3! + 1) \times (3! + 1)) = 8,621$ *attack scenario*. This is a large number of cyberattack scenarios; however, most of them are irrelevant. The pre-and-post conditions of CPN attacks must match the CPN PIT protocol for integration.

Algorithm 4 generates cyberattack scenarios. It performs a brute-force search of the CPN PIT protocol to integrate CPN attacks. For each CPN attack, it checks the CPN nodes of the PIT protocol to identify where the CPN attack's pre- and postconditions are satisfied–these are the exact points where the CPN attack can be integrated. The CPN attack satisfies the postcondition of the corresponding CPN node, demonstrating an alteration for any token that passes through these nodes. This is how the various cyberattack scenarios are constructed. Algorithm 4 attaches these CPN attacks to the CPN of the PIT protocol based on pre-and postconditions. For instance, the Tampering CPN attack on the *Recipient OTP* requires an *OTP* token that includes the encrypted OTP needed to unlock the *Device*. The Tampering CPN attack intercepts this token, changes the *OTP* code, and passes it back to the *Recipient*, leaving the *Recipient* unable to unlock the *Device* because the verification of *OTP* fails.

The 6 CPN attacks, shown in Fig. 13, are integrated into the CPN of the PIT protocol to demonstrate various cyberattack scenarios. We investigate 6 cyberattack scenarios, shown in Table 1, focusing on the communication channel between *Recipient*, *HRoT*, and *Server*, *Server* availability, and *Recipient* impersonation. The first, second, and third cyberattack scenarios demonstrate one attack each, examining the communication channels as *Recipient-Server*, *Server-Recipient*, and *Server-HRoT*, respectively. The fourth cyberattack scenario also demonstrates one attack, but it examines the situation in which a malicious *Recipient* tries to unlock the device by entering an incorrect *OTP*. The fifth and sixth cyberattack scenarios utilize a combination of attacks. The fifth cyberattack scenario includes spoofing and tampering attacks targeting the communication channel between *HRoT* and the *Server* while the sixth cyberattack scenario employs Elevation of Privilege and DoS attacks against the *Server* trying to render it unavailable.

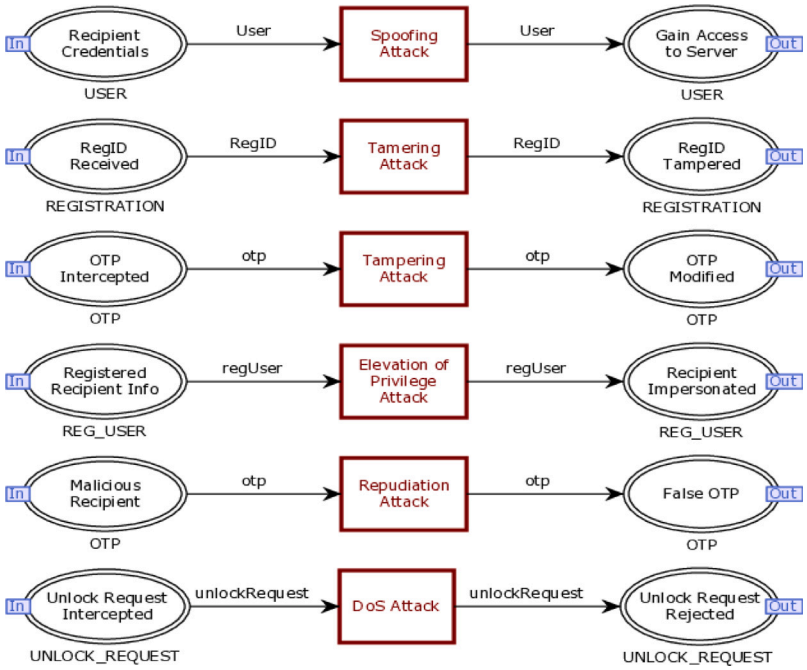

Fig. 13. Coloured petri nets of attacks.

Algorithm 4: CPN PIT Protocol & CPN Attacks Integration

Input : $CPN\_Attack = \langle P, T, A, \Sigma, V, C, G, E, I \rangle$
Input : $CPN\_Protocol = \langle P, T, A, \Sigma, V, C, G, E, I \rangle$
Output: $Attack\_Scenarios = \langle P, T, A, \Sigma, V, C, G, E, I \rangle$

/* for each transition in CPN-Attack, get pre-and-post places */
foreach $t_i \in CPN\_Attack.T$ **do**
  $p^{pre}_{attack} \leftarrow get(p^{pre}_i, t_i)$;
  $p^{post}_{attack} \leftarrow get(p^{post}_i, t_i)$;
  /* iterate over CPN-Protocol transitions, and */
  /* find match with pre-and-post places */
  foreach $t_j \in CPN\_Protocol.T$ **do**
    $p^{pre}_{protocol} \leftarrow get(p^{pre}_j, t_j)$;
    $p^{post}_{protocol} \leftarrow get(p^{post}_j, t_j)$;
    /* if match found, attach the attack, and store the new CPN-Protocol as an attack scenario */
    if $p^{pre}_{attack} = p^{pre}_{protocol}$ AND $p^{post}_{attack} = p^{post}_{protocol}$ **then**
      $createInPort(P \leftarrow AddPlace(p^{inPort}_j))$;
      $createOutPort(P \leftarrow AddPlace(p^{outPort}_j))$;
      $connect(CPN\_Protocol, CPN\_Attack, p^{inPort}_j, t_j, p^{outPort}_j)$;
      $CPN\_Scenarios \leftarrow AddAttackScenario(CPN\_Protocol)$;
    end
  end
end
return $CPN\_Scenarios$

5.2. Security analysis of PIT protocol via reachability

The six cyberattack scenarios, presented in Table 1, were executed by CPNTools (Ratzer et al., 2003). For each cyberattack scenario, the state space and reachability were analyzed.

State Space Analysis: The analysis examines the state space for each attack connected to the CPN of the PIT protocol.

Table 2 reports that all CPNs in cyberattack scenarios are strongly connected components (SCC). This indicates that the state model generated by the CPN is acyclic; it does not have infinite loops. It also reports many dead markings and dead transitions. Dead markings are states where the CPN is terminated. The dead transitions are those transitions that never occur (a token may not visit them). Dead markings and dead transitions are considered for verification. Verifying the dead markings signifies the values of the PIT protocol's parameters where the PIT protocol is terminated, whereas verifying the dead transitions signifies why the PIT protocol did not execute a specific functionality. The state space of the first attack scenario reports 3 dead markings ([20,21,31]) and 2 dead transitions. The second attack scenario shows 4 dead markings ([20,21,25,31]) and 2 dead transitions. The third attack scenario shows 4 dead markings ([18,19,21,24]) and 9 dead transitions. The fourth attack scenario shows 2 dead markings ([28,31]) and 1 dead transition. This dead transition instance is reported as "$UNLOCK DeviceUser'Unlock\_the\_Device$ 1". The fifth attack scenario reports 2 dead markings ([18,21]) and 3 dead transitions. The sixth attack scenario reports 3 dead markings ([29,32,33]) and 2 dead transitions.

Reachability Analysis: For each attack scenario, we backtrack through the execution sequence and locate the states where the attacks occur. While backtracking, we verify the change of PIT protocol's states to inspect the reasons for dead transitions and what state values the sequence terminates with at dead markings.

The first and second cyberattack scenarios are similar, although the attack types differ: one is a spoofing attack, and the other is a tampering attack. This is because attacks targeted *Server–Recipient* and *Recipient–Server* channels, spoofing and tampering with the $RegIDs$. Upon closer examination of these dead markings and transitions, it becomes evident that these attacks were thwarted. The reason? The spoofed and tampered $RegIDs$ were rendered invalid once the *Server* decrypted it and compared to the stored $RegID$. Although the *Recipient*'s original access request was incomplete, the PIT protocol was able to defend against these attacks. The third cyberattack scenario failed to bypass the $RegID$ verification. This cyberattack scenario reports 9 dead transitions because the CPN of the unlocking phase did not reach the

Table 1
PIT Protocol Attack Scenarios.

Scenario	Target Entity	Attack	Expected Impact
1	Recipient–Server Channel	Spoofing Recipient credentials	Impersonate Recipient, gain access to the Server
2	Server–Recipient Channel	Spoofing OTPs	Know OTP to launch MitM attack, and compromise HRoT
3	Server–HRoT Channel	Tampering RegIDs	Disrupt the unlock process
4	Recipient–HRoT Entry	Repudiation of OTPs	Impersonate Recipient and try to gain access to HRoT
5	HRoT–Server Channel	Spoofing and Tampering OTPs	Know OTP to launch MitM attack, and disrupt the unlock process
6	Server	Elevation of Privilege and DoS	Using Recipient credentials, gain unauthorized privileges on the Server, intercept unlocking requests and dump the responses, and rendering Recipient unable to unlock the Device

Table 2
State Space Report with Attack Scenarios.

Entity & Attack Scenario	State Space		SCC Graph		Status
(1) Recipient–Server Channel Spoofing Recipient Credentials	#Node 31	#Arcs 30	#Node 31	#Arcs 30	Full
	Dead Markings [20,21,31]		#Dead Transition 2		Live Transition None
(2) Server–Recipient ChannelSpoofing OTPs	#Node 31	#Arcs 30	#Node 31	#Arcs 30	Full
	Dead Markings [20,21,25,31]		#Dead Transition2		Live Transition None
(3) Server–HRoT ChannelTampering RegIDs	#Node 24	#Arcs 23	#Node 24	#Arcs 23	Full
	Dead Markings [18,19,21,24]		#Dead Transition9		Live Transition None
(4) Recipient–HRoT EntryRepudiation of OTPs	#Node 31	#Arcs 30	#Node 31	#Arcs 30	Full
	Dead Markings [28,31]		#Dead Transition1		Live Transition None
(5) HRoT–Server ChannelSpoofing and Tampering OTPs	#Node 21	#Arcs 23	#Node 21	#Arcs 23	Full
	Dead Markings [18,21]		#Dead Transition3		Live Transition None
(6) Server Elevation ofPrivilege and DoS	#Node 33	#Arcs 38	#Node 33	#Arcs 38	Full
	Dead Markings [29,32,33]		#Dead Transition2		Live Transition None

states where *OTP* verification occurs. Thus, this cyberattack scenario was thwarted in the early stages. The fourth cyberattack scenario failed because the CPN repudiation attack entered an incorrect *OTP* trying to unlock the device. The *HRoT* compares the *OTP* entered with the *OTP* received from the *Server*, which was mismatched. Thus, this cyberattack scenario was also thwarted. Similarly, investigating the dead markings and transitions generated from the fifth scenario indicate that the *HRoT* and *Server* successfully thwarted the spoofing and tampering attacks on the *OTPs* once they decrypted it and compared it to the original *OTP*. The sixth attack scenario successfully gained access to the *Server* and effectively incapacitated the *Server*, causing disruption to the PIT protocol operations. This attack scenario indicates that the *Server* must have strong authentication and authorization mechanisms to be secure.

Security of the PIT Protocol: We examine 6 cyberattack scenarios to attempt to violate the correctness properties of the PIT protocol outlined in Section 2.1. Specifically, we focus on properties 1, 2, and 3 since the violation of these three properties would compromise the integrity and availability of the PIT protocol. We want to investigate whether these properties hold true in relation to these cyberattack scenarios. This analysis will strengthen our confidence in the proven security of the PIT protocol.

Scenario 1 - Recipient–Server Channel Spoofing Recipient Credentials. This attack scenario impersonates the *Recipient* by spoofing $RegID$ to violate the first correctness property: *"If Recipient is malicious and the Device is incorrect, the PIT protocol prevents unlocking the Device"*. The PIT protocol verified the *Recipient*'s credentials and Device's $RegID$. Since the *Recipient* is not registered and is malicious, the process does not proceed to verify whether the *Device* is correct or incorrect. As a result, the PIT protocol rejected the request to unlock the *Device*.

Scenario 2 - Server–Recipient Channel Spoofing OTPs. This attack scenario spoofs *OTPs* to compromise *HRoT*. It attempts to violate the third correctness property: *"If Recipient is malicious and the Device is correct, the PIT protocol prevents unlocking the Device"*. The PIT protocol decrypts the *OTPs* and compares with the *OTP* issued by *HRoT*. The PIT protocol found that these OTPs did not match; therefore, the request to unlock the *Device* is failed.

Scenario 3 - Server–HRoT Channel Tampering RegIDs. This attack scenario involves tampering with $RegIDs$ to violate the second correctness property: *"If Recipient is legitimate and the Device is incorrect, the PIT protocol prevents unlocking the Device"*. The attacker managed to bypass registration verification by obtaining $RegIDs$ and attempted to unlock *Device*. However, the PIT protocol successfully rejected the request to unlock *Device* using the OTP mechanism.

Scenario 4 - Recipient–HRoT Entry Repudiation of OTPs. This attack scenario impersonates the *Recipient* by physically entering *OTP* into the *Device* to unlock it. Similarly to Scenario 2, it attempts to violate the third correctness property: *"If Recipient is malicious and the Device is correct, the PIT protocol prevents unlocking the Device"*. With a

limited number of attempts, the PIT protocol found that these OTPs did not match; therefore, the request to unlock *Device* is rejected.

Scenario 5 - HRoT–Server Channel Spoofing and Tampering OTPs. This attack scenario intercepts the channel between *HRoT* and *Server* and tampering with *OTP* to unlock the *Device*. The scenario attempts to violate the third correctness property: *"If Recipient is malicious and the Device is correct, the PIT protocol prevents unlocking the Device"*. The PIT protocol uses robust encryption and decryption algorithms for OTP verification; therefore, the tampered OTP used to request to unlock *Device* is failed.

Scenario 6 - Server Elevation of Privilege and DoS. This attack scenario attempts to compromise the *Server* by applying elevation of privilege and DoS attacks to render it unavailable. It investigates whether *Recipient* can authenticate and unlock *Device* without *Server* intervention. The scenario attempts to violate the third correctness property: *"If Recipient is malicious and the Device is correct, the PIT protocol prevents unlocking the Device"*. While protecting *Server* is beyond the scope of the PIT protocol, as the goal of the protocol is to protect the *Device*, it is important to note that *Server* plays a crucial role in authenticating both *Recipient* and *Device*. Therefore, this attack scenario was unsuccessful because the PIT protocol failed to deliver OTP issued by the *HRoT* to the *Server*. Moreover, it could not verify that the *Recipient* receiving OTP from the *Server* to unlock the *Device* in the absence of the *Server*.

The results of the formal verification have proven the security conditions of the PIT protocol property. The PIT protocol is robustly safeguarded against attacks, and the Device remains "locked" until a secure connection and authentication are established. It prevents spoofing and tampering attacks that occur in communication channels between the *Recipient*, *Server*, and *HRoT*. The PIT protocol uses advanced encryption and decryption algorithms ensuring authenticity; the *Recipient* is authenticated to *Server*, and both *Recipient* and *Device* authenticate each other. It protects the *Device* against impersonation and data integrity cyberattack. Nonetheless, the PIT protocol is not immune to all threats. Cyberattacks targeting the *Server*, such as DoS attacks, can compromise the protocol, potentially preventing a legitimate *Recipient* from unlocking the correct *Device*. This vulnerability underscores the need to address this issue. Although *Device* remains locked and secure even in the event of a *Server* compromise, it is important to protect all sites of the PIT protocol for its correct execution.

6. Implementation of PIT protocol

The implementation of the PIT protocol (Podder et al., 2024) has been conceived by Microsoft as Project Cerberus, is a framework for HRoT that is compliant with the NIST 800-193 standards, with an unclonable identity. This platform enforces a secure boot process for *Device* firmware and provides a secure mechanism to attest to the state of the *Device* firmware.

The implementation of PIT protocol within Project Cerberus is distributed as a library called "*pit*" consisting of several Application Programming Interfaces (APIs). These APIs encapsulate the various functionalities that the *HRoT* processor, a key component, provides to lock and unlock the *Device*'s BIOS. The design of these APIs is guided by the principle of enabling the PIT protocol to lock and unlock the BIOS securely and efficiently. The following list of APIs describes the functionalities of the PIT protocol.

6.1. PIT protocol's core library

The library is built upon existing functions provided by Project Cerberus. According to the architecture of the PIT protocol, the APIs of "*pit*" are illustrated in Tables 3 and 4.

Figs. 14 and 15 describe the sequence of function calls required to implement the PIT protocol. In the *Server* implementation, we leverage the Python cryptography library, employing three key modules: Elliptic Curve Cryptography (ECC), Serialization, and Advanced Encryption Standard-Galois/Counter Mode (AES-GCM) (Podder and Barai, 2021). The ECC module generates a public–private key pair for the *Server*. It furnishes a method to perform the Elliptic-curve Diffie–Hellman (ECDH) protocol, enabling the generation of a shared secret using another party's public key. This module is indispensable to our implementation, although any library that supports the generation of an elliptic curve key pair and the ECDH protocol — compliant with NIST standards (Barker, 2016) — would suffice.

The serialization module is tasked with key distribution and reception. Before the *Server*'s public key transmission to *HRoT*, it is encoded into DER format via the serialization module. Similarly, the *HRoT* public key is processed by DER serialization before being sent to the *Server*. The *Server* uses the serialization module to convert the DER-encoded public key from *HRoT* into a format compatible with the cryptography library, precisely an EC key. The libraries provide key serialization following the NIST standards. The AES-GCM module is responsible for data encryption and decryption. AES-GCM is the chosen form of AES owing to its native support in Project Cerberus. It is coupled with the shared secret derived from ECDH; the AES-GCM module is the key for encryption and decryption, ensuring secure and coherent communication between *Server* and *HRoT*. For *Devices* requiring Project Cerberus functionality, the core set of source code delivers a suite of foundational features that can be integrated and ported. The provided code, largely *Device*-agnostic, defines the required abstraction layers. Therefore, we tailor Project Cerberus by introducing a new library and APIs that include the lock and unlock mechanisms for BIOS. Detailed documentation on the APIs and "*pit*" library is publicly available on GitHub (Podder and Sovereign, 2023).

6.2. Evaluation of PIT protocol

Our primary focus with this evaluation was to analyze the performance of the PIT protocol framework with the library. We developed various experimental test scenarios to evaluate our framework with the library's performance. We ran our experiments on 2 virtual Linux *Server*s. The client side (assumed as *HRoT*) has a 5000 MHz 12th Gen Intel(R) Core(TM) i7-12700K processor, x86_64 architecture, 20 CPUs, and *Server* in Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60 GHz, x86_64 architecture, 12 CPU(s).

To evaluate whether the PIT protocol framework is working as it is supposed to, we used *Server*-level Project Cerberus with the "*pit*" library with a C socket as client and a *Server* implemented in Python. As designed and expected, our protocol delivers the expected results, with successful operations observed in both *HRoT* with the PIT protocol and the *Server*.

The *Device* (with an integrated BIOS) initiates a locked state as the *Server*, equipped with its own distinct set of ECC key pairs, establishes a connection with *HRoT*. Following the successful exchange of pairs of ECC key pairs between *HRoT* and the *Server*, the secret key computation starts. We designed a simple two-step verification protocol for the logging in. Once a *Recipient* logs in with credentials, an unlock request prompts on the *Server*. The *Server* then commences communication with the *HRoT*. The unlocking process initiates when the *Server* transmits the encrypted product registration ID (*RegIDs*) to *HRoT*. Subsequently, *HRoT* decrypts and validates this product registration ID, and the output is displayed in the *HRoT* terminal. *HRoT* generates a one-time password *OTP* upon successful validation of the product registration ID. This *OTP* is encrypted using AES-GCM with the secret key (S) and dispatched to the *Server*, which then forwards this encrypted *OTP* (OTP_S) to the *Recipient*'s email address. Upon receiving the *OTPs*, the *Recipient* inputs it into the *HRoT* terminal. *HRoT* then decrypts this *OTPs* utilizing the secret key (S) and performs a validation check. If validation is successful, *HRoT* unlocks *Device*.

In addition to the evaluation of the "*pit*" library, we conducted extensive testing ("Compatibility Testing") to validate their compatibility with different *Server* libraries that are used for the generation

Table 3
Description of PIT Protocol core libraries (Cryptographic APIs).

Directory	Function/API	Description
PIT Protocol's Cryptographic APIs		
pit_crypto	`pit_keygenstate()`	This function is responsible for generating the ECC key pair for *HRoT*. It loads the length of the key in *key_length*, initializes private key in *privkey*, public key in *pubkey*, and the state of the *HRoT* in *state* parameter.
	`pit_secretkey()`	It takes ECC private key from *HRoT* and *Server*, computes the secret key and loads in *secret* parameter.
	`pit_encryption()`	It encrypts a message using a secret key. This function takes secret key, message and use AES-GCM method to encrypt the message and loads into *ciphertext* parameter.
	`pit_decryption()`	This function takes encrypted message (*ciphertext*), secret key (*secret*) as input, decrypts it and loads the message to the provided *plaintext* buffer.
	`generateDSA()`	Generates a Digital Signature Algorithm (DSA) key pair.
	`pit_OTPgen()`	This function generates a random string representing *OTP*. Additionally, encrypts that *OTP* using the secret key (*secret*) as key for the AES-GCM and loads in *encOTP* parameter.
	`pit_OTP _validation()`	This function validates the encrypted OTP (*encOTP*). It does this by taking encrypted *OTP* (OTP_S) as input, decrypting it, and comparing it against the original OTP (*OTP*). If valid, the function returns 1 and the parameter *result* will hold *true* otherwise *false*.

Table 4
Description of PIT Protocol core libraries (Main and Communication APIs).

Directory	Function/API	Description
PIT Protocol's Main APIs		
pit	`pit_Lock()`	This function uses the internal pit_keygenstate(), keyexchangestate(), pit_secretkey() to perform all the operations needed to lock the BIOS and loads the secret key in a 32-byte empty array (*secret*).
	`pit_Unlock()`	Do all the unlocking operations. These operations generate OTP, encrypt it, send to *Server* and validate the OTP.
PIT Protocol Communication APIs		
pit_client	`pit_connect()`	It initiates a connection to designated *Server*. It takes the port address of the *Server* as a input and returns an integer pointing to the file descriptor (socket) which can be used to send/receive from the *Server*.
	`keyexchangestate()`	Exchange the public key of *HRoT* and *Server*. On success, keyexchangestate will initialize *pubkey_cli* with the *HRoT*'s public key and load the *pubkey_serv* variable with a public key received from the *Server*.
	`send_unlock_info()`	This function sends the encrypted OTP (*encOTP*) to the *Server* which will be later sent to *Recipient*.
	`receive_product _info()`	This function receives the product information from the *Server* and assigns it to some parameters, such as encrypted registration ID (*EncryptedProductID*), tag (*EncryptedProductIDTag*), registration ID size (*ProductIDSize*), an initialization vector (IV) used for encryption (*aes_iv*), and size (in bytes) of the vector in (*aes_iv_size*).

of ECC key pairs and shared secret key. Different *Server* environments employ libraries for the ECC key pair generation and shared secret key computation, each with intricacies and idiosyncrasies. Confirming that the "*pit*" library interacts seamlessly with these different *Server* libraries without any interoperability issues was crucial. To achieve this, we conducted a series of interoperability tests in multiple *Server* environments, each employing different libraries for key pair generation and shared secret key computation. We tested with commonly used libraries, such as OpenSSL (Viega et al., 2002), libsodium (Consortium, 2013), and Cryptlib (Gutmann, 2008), which are widely accepted for their robustness and compliance with industry standards. In each case, we confirm that the "*pit*" library used in *HRoT* efficiently computes and exchanges ECC key pairs and shared secret keys without compatibility issues. We also ensure that the computed keys adhere to the relevant NIST standards and that the elliptic-curve cryptography functionality is up to the mark. Furthermore, we have validated that the PIT library correctly interprets the serialized keys received from the *Server* and successfully executes the Elliptic Curve Diffie–Hellman (ECDH) operation to generate a shared secret key. We have also verified the successful encryption and decryption of the data using the derived shared secret key. This extensive testing has validated that the "*pit*" library is compatible with a wide range of libraries for generating ECC key pairs and shared secret keys, thus enhancing our protocol's universal applicability and adaptability.

To validate the compatibility of the "*pit*" library, we design a proof of concept incorporating an OpenSSL-based *Server*. The objective is to demonstrate the universal capabilities of our designed library in handling operations regardless of the library types. By executing our model, *HRoT* transmits the public key to the *Server*, encoded in Distinguished Encoding Rules (DER) format. The OpenSSL *Server* effectively interprets the public key that *HRoT* delivers. The DER format public key is subsequently outputted in the *Server* terminal, further validating the successful exchange of cryptographic keys between *HRoT* and the *Server*.

Further investigation shows that OpenSSL is proficient in interpreting both public and private keys generated by the *Server*. This empirical

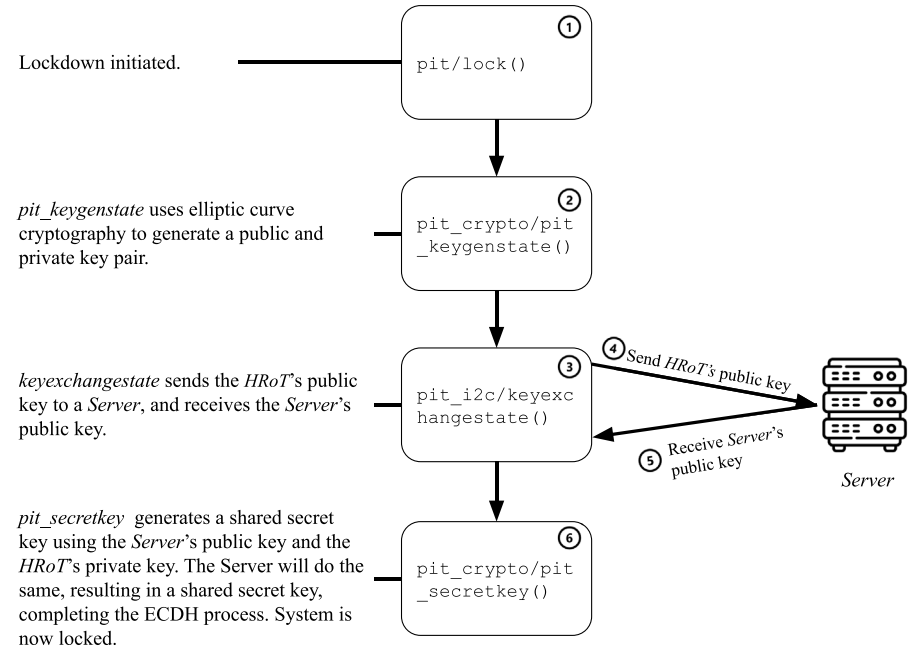

Fig. 14. Function call sequence in Locked State of "*pit*".

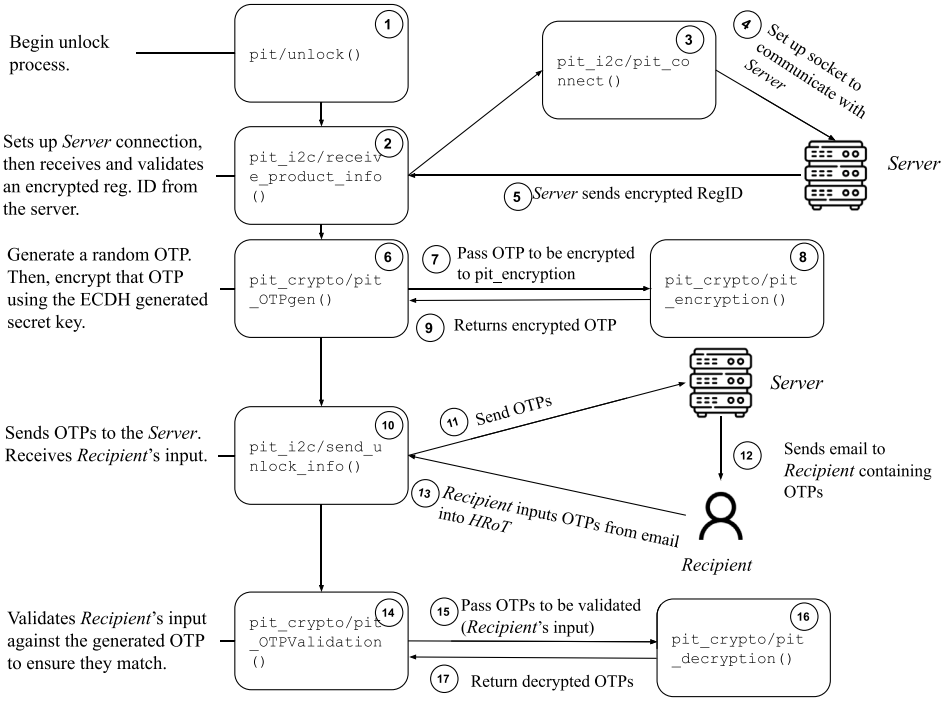

Fig. 15. Function call sequence in Unlocked State of "*pit*".

evidence substantiates that the cryptographic library, as implemented in our framework, is compatible with OpenSSL and, by extension, compliant with the National Institute of Standards and Technology (NIST) standards for cryptographic algorithms. Therefore, our methodology exhibits a broad spectrum of interoperability, ensuring compatibility with cryptographic systems that adhere to NIST standards. This evidence proves that using OpenSSL in conjunction with our Python *Server* produces identical secret keys. This demonstrates the robustness and reliability of our key generation process and unequivocally confirms the compatibility of our implementation with systems that conform to NIST standards. As such, we can ensure the broader applicability of our system, facilitating secure and effective communication between various cryptographic frameworks that adhere to these industry-accepted norms.

Additionally, we thoroughly tested the protocol's exceptions and failure-handling capability. We exercise various test scenarios.

Table 5 shows 4 instances of testing scenarios. These 4 test scenarios were specifically designed to validate the unlocking phases, ensuring the protocol's robustness. We also executed test scenarios where the *Device* is locked by the *Recipient* and shipped to another *Recipient*; the process started from the phase of the key_gen_state to the unlocking_state. Through these testing scenarios, we addressed cases of false positives, ensuring that legitimate users are not permanently

Table 5
Protocol's exceptions and failures handling capability in various test scenarios.

Test Scenarios	Description	Handling	State
Server unavailability	The *Server* loses connections during locking or unlocking	The program will wait until the timer specified time. If the connection is not back, it will revert back to the previous state and throw a timeout exception.	Lock state
RegID validation fails	The encrypted $RegID_S$ fails to validate due to wrong *RegID* or empty.	If the encrypted *RegIDs* cannot be validated, the program throws a *RegID* validation failure error and reverts back to the lock state.	Lock state
OTP validation fails	The encrypted *OTPs* fails to validate due to wrong *OTP* or empty generated by *HRoT*.	If the encrypted *OTPs* cannot be validated, the program throws a *OTP* validation failure error and reverts back to OTP_gen state.	OTP_gen state
Unable to send *OTPs* to *Recipient* email	The encrypted *OTPs* fails to deliver to the *Recipient* due to wrong connectivity issues.	If the encrypted OTP_S is not available to the *Recipient*, the *HRoT* will wait until the specified timer expires, then throw a timeout exception and revert back to the OTP_gen state.	OTP_gen state

blocked due to protocol failures or unexpected exceptions. To support this, our implementation incorporates detailed states within the code, facilitating prompt debugging, accurate diagnostics, and rapid recovery whenever exceptions or failures occur.

We have successfully modified the Project Cerberus embedded framework for *HRoT*, a system equipped to secure the product during transit by locking the *Device* (with BIOS). PIT protocol boasts the cryptographic capabilities required by a microcontroller or microprocessor for BIOS locking. Our work, including the "*pit*" library, is publicly available on our GitHub repository (Podder and Sovereign, 2023). Moreover, a comprehensive set of instructions for operating the PIT protocol with Project Cerberus has been provided, further promoting the solution's accessibility and usability.

7. Related work

Security researchers have demonstrated several attacks against conventional BIOS and EFI/UEFI firmware under laboratory conditions (Cooper et al., 2011; Fuchs et al., 2016; Vasselle et al., 2019; Haken, 2015; Jack, 2010; Podder et al., 2025).

Sacco and Ortega (2009) discuss the topic of BIOS infections and describe a proof-of-concept demonstration of a persistent BIOS malware infection. The authors claim that BIOS infections are challenging to detect and eradicate as they reside in a computer's memory and are inaccessible to standard antivirus or anti-malware software. Wojtczuk and Tereshkin (2009) conduct research on the security of Intel BIOS and described several vulnerabilities and attack vectors that can be utilized to attack the BIOS, such as exploiting firmware vulnerabilities, accessing the BIOS via hardware debugging tools, and employing software-based assaults such as buffer overflow.

Malware at the firmware level has been more common in embedded systems. Embleton et al. (2008) introduce a new type of malware known as System Management Mode (SMM) rootkits. The authors explain the nature of SMM, its role in computer hardware, and the vulnerabilities that can be used to access SMM. Cui et al. (2013) focus on firmware modification attacks and their exploitation. Using the HP-RFU vulnerability in LaserJet printers as a case study, the authors demonstrate the development of a proof-of-concept printer malware capable of network reconnaissance, data exfiltration, and propagation to other devices. They highlight the widespread nature of vulnerable embedded devices and the challenges in patching them, with only a small percentage of the vulnerable population being patched. The paper also emphasizes the limitations of firmware update signing and identifies vulnerabilities in third-party libraries found in firmware images. Overall, the findings underscore the need for effective host-based defense mechanisms to protect vulnerable embedded systems.

Numerous firmware-level cyberattack incidents have been reported and studied. PsycoB0t (Maassen, 2009), a notable router botnet, infiltrated the firmware of about 85,000 DD-WRT home routers, turning them into instruments for conducting severe network-disrupting Distributed Denial of Service (DDoS) attacks. Barnaby Jack demonstrates the unauthorized cash extraction from ATMs by modifying their firmware, a technique infamously known as "jackpotting" (Jack, 2010). Charlie Miller uncovers severe risks within the firmware of certain Apple laptop batteries that could lead to malfunctions or overheating cite miller2011battery. Employing PostScript (Costin, 2011), a ubiquitous language in electronic and desktop publishing, Costin showcases the susceptibility of certain Lexmark printers to memory inspection and arbitrary modifications, which can lead to exposure of sensitive data or disruption of device functionality. Kevin Fu's groundbreaking work in medical device security highlights the perilous reality of exploiting embedded devices (Hanna et al., 2011), with his real-world attacks on an implantable cardioverter defibrillator and an automated external defibrillator illuminating these life-threatening vulnerabilities. These instances underscore the crucial need for robust firmware security across diverse devices, as firmware forms the base code controlling hardware, making successful infiltration by an attacker profoundly dangerous. It is imperative that robust firmware security measures are implemented to mitigate these risks.

The integrity of the system BIOS is critical for ensuring the security and functionality of computer systems, particularly during their transit through the supply chain. Unauthorized modifications, such as malicious firmware updates and modifications, pose significant threats (Schmidt et al., 2016). Malicious actors exploit vulnerabilities or weak security measures to alter the BIOS, introducing malware or Trojans that can lead to data breaches, device malfunctions, or system takeovers (Basnight et al., 2013). Man-in-the-middle (MITM) attacks exacerbate these threats (Conti et al., 2016) and supply chain attacks (Miller, 2013), which intercept or tamper with firmware updates or implant malware, respectively. Such attacks exploit the low-level operation of firmware, making detection and mitigation difficult, and represent a persistent security challenge (Cooper et al., 2011; Fuchs et al., 2016; Vasselle et al., 2019). Hardware-based trusted computing utilizes Trusted Execution Environments (TEE) and secure elements like the Trusted Platform Module (TPM) (Mitchell, 2005) to enhance security. TPMs, which have been included in computers for over a decade, establish a root of trust in a secure cryptographic core, protecting keys and credentials even if the OS is compromised. The latest standard, TPM 2.0, is present in most modern computers and supports various elliptic curve signature schemes, essential for certain core security services (Moghimi et al., 2020). However, vulnerabilities like CVE-2020-10713 in the GRUB2 bootloader and others in bootloaders from Eurosoft, New Horizon Datasys, and CryptWare have shown that attackers can bypass Secure Boot to execute malicious code. These vulnerabilities can be mitigated by blocklisting the affected bootloaders in the UEFI Secure Boot Forbidden Signature Database (DBX), which can be updated via UEFI firmware or Windows updates. Beyond bootloaders, attackers can target more profound UEFI components, exploiting specific vulnerabilities for targeted attacks. Recent research has

uncovered numerous high-impact vulnerabilities (CVE-2022-28858, CVE-2022-36372, CVE-2022-32579, CVE-2022-27493 and CVE-2022-33209, CVE-2022-23930, CVE-2022-31644, CVE-2022-31645, CVE-2022-31646, CVE-2022-31640 and CVE-2022-31641) in UEFI firmware components, indicating an ongoing risk despite advancements in UEFI security technologies (Constantin, 2022).

The NIST SP 800-193 guidelines (Regenscheid, 2018) offers comprehensive recommendations for protecting firmware from unauthorized alterations. However, these guidelines have limitations when an attacker gains physical access to a device during transit. They lack robust post-tamper detection mechanisms after an attack and fail to address physical security situations specific to the device during transit. In diverse systems, inconsistency in the implementation of security protocols across different devices and the complexity of global supply chains enlarge the attack surface, increasing vulnerabilities. Moreover, local update processes and key management challenges further elevate the risks, particularly in environments with a variety of device architectures. Khalid et al. (2013) propose a robust solution for ensuring software integrity during boot-up, but with certain drawbacks. The approach mitigates software-based attacks, offering limited defense against physical, side-channel, and fault-injection attacks. It also introduces hardware overhead and increases boot time by approximately 25 Remote attestation offers potential for securing firmware, but as Coker et al. (2011) note, existing attestation systems face several limitations. One fundamental issue is the inflexibility of many attestation mechanisms, which are often tailored to specific use cases and may not scale across diverse environments. Another challenge is ensuring the trustworthiness of the attestation process when it spans multiple organizations or components, which can lead to incomplete or unverifiable claims about the system's security state.

Coloured Petri Nets (CPN) is widely used for security protocol verification, effectively modeling and analyzing protocols. Zhang and Miao (2024) highlight CPN's ability to execute and detect new attacks, applying it to the Tatebayashi, Matsuzaki, and Newman (TMN) protocol in mobile communication. Using the Dolev-Yao attacker model, they identified vulnerabilities and compared CPN with tools like Scyther, AVISPA, Tamarin, and ProVerif, noting challenges like state space explosion. Similarly, Bhurke and Kazi (2021) reviewed formal methods in Industrial Control Systems (ICS), focusing on CPN's use in analyzing the HART-IP protocol against attacks like Denial of Service (DoS), Man-in-the-Middle (MitM) and Replay Attacks, demonstrating its versatility in industrial environments.

CPN have also been used to verify IoT and industrial protocols. Ahmad et al. (2023) applied CPN to verify the security of the Routing Protocol for Low power and lossy networks (RPL) in IoT systems, covering both secure and non-secure modes. Feng et al. (2024) used CPN to analyze the ISA100.11a protocol, widely adopted in industrial automation, revealing vulnerabilities like device ID spoofing and replay attacks. Similarly, Moradi and Jahangir (2024) modeled Precision Time Protocol (PTP) using CPN to detect vulnerabilities related to Time-Delay Attacks (TDAs), demonstrating its effectiveness in securing time-sensitive distributed systems.

8. Conclusion & future work

The Protection In Transit (PIT) protocol safeguards computing devices during transit, addressing a critical security gap in the firmware. It executes a lock and unlock mechanism to secure the BIOS and verifies the user's authenticity before the BIOS boots up, ensuring that the device's firmware remains protected throughout transit.

We designed, verified, and implemented the PIT protocol intensively. The specifications and correctness properties of the PIT protocol are formally described, including the lock and unlock mechanism, the encryption and decryption scheme, and the two-factor authentication technique. Its components interaction and data flow are also described and visualized using UML sequence diagrams and data flow diagrams. We used formal methods to verify the correctness of the PIT protocol as it prevents unauthorized firmware modifications. We have proven this correctness property of the PIT protocol using Coloured Petri Nets (CPN). We also used the STRIDE threat modeling framework to demonstrate various cyberattack scenarios. This comprehensive verification analyzes the PIT protocol's security, assuring it is resilient against cyberattacks during device transit. The PIT protocol has been implemented through an open source embedded framework, Project Cerberus, which Microsoft conceived. The APIs and core libraries implementing the PIT protocol's functionalities are listed and described. The PIT protocol library ("*pit*") is publicly available to the community.

Once the device is securely received, our consideration focuses on securing the remote firmware update procedure. Remote firmware updates (RFUs) are essential for maintaining and enhancing the functionality and security of devices, especially those in distributed and IoT environments. Therefore, our research will extend to the development of a Secure Remote Firmware Update Protocol (S-RFUP) to ensure the integrity, authenticity, and reliability of updates. Furthermore, we will continue to work on establishing standards for interoperability between different vendors and platforms, ensuring that the PIT protocol can be widely adopted and integrated into various hardware and firmware ecosystems.

CRediT authorship contribution statement

Rakesh Podder: Writing – review & editing, Writing – original draft, Validation, Methodology, Formal analysis, Conceptualization. **Mahmoud Abdelgawad:** Writing – review & editing, Writing – original draft, Validation, Methodology, Formal analysis, Conceptualization. **Indrakshi Ray:** Writing – review & editing, Supervision, Funding acquisition, Formal analysis, Conceptualization. **Indrajit Ray:** Writing – review & editing, Supervision, Funding acquisition, Conceptualization. **Madhan Santharam:** Resources, Project administration. **Stefano Righi:** Resources, Project administration.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Mahmoud Abdelgawad reports financial support was provided by U.S. National Science Foundation. Rakesh Podder reports financial support was provided by AMI US Holdings Inc. Madhan Santharam reports a relationship with AMI US Holdings Inc. that includes: employment. Stefano Righi reports a relationship with AMI US Holdings Inc. that includes: employment. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by the U.S. National Science Foundation under Grant No. CNS 1822118 and CNS 2226232, Award Numbers DMS 2123761, the member partners of the NSF IUCRC Center for Cyber Security Analytics and Automation – AMI, NewPush, Cyber Risk Research, NIST and ARL – the State of Colorado (grant #SB 18-086), Colorado State University, and by NIST, United States under Award No. 60NANB23D152. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, or other organizations and agencies.

Data availability

Podder, R., Sovereign, J., 2023. Project Cerberus with pit protocol. URL: https://github.com/cryptoknight13/Project-Cerberus.

References

Ahmad, F., Chaudhry, M.T., Jamal, M.H., Sohail, M.A., Gavilanes, D., Vergara, M.M., Ashraf, I., 2023. Formal modeling and analysis of security schemes of RPL protocol using colored Petri nets. Plos One 18 (8), e0285700. http://dx.doi.org/10.1371/journal.pone.0285700.

Barker, E., 2016. Guideline for using cryptographic standards in the federal government: Cryptographic mechanisms. NIST Spec. Publ. 800–175B.

Basnight, Z., Butts, J., Lopez Jr., J., Dube, T., 2013. Firmware modification attacks on programmable logic controllers. Int. J. Crit. Infrastruct. Prot. 6 (2), 76–84. http://dx.doi.org/10.1016/j.ijcip.2013.04.004.

Bhurke, A.U., Kazi, F., 2021. Methods of formal analysis for ICS protocols and HART-IP CPN modelling. In: Proceedings of the Asian Conference on Innovation in Technology. IEEE, PUNE, India, pp. 1–7.

Coker, G., Guttman, J., Loscocco, P., Herzog, A., Millen, J., O'Hanlon, B., Ramsdell, J., Segall, A., Sheehy, J., Sniffen, B., 2011. Principles of remote attestation. Int. J. Inf. Secur. 10, 63–81.

Consortium, I.S., 2013. Libsodium documentation: Introduction. URL: https://libsodium.gitbook.io/doc/.

Constantin, L., 2022. New exploits can bypass secure boot and modern UEFI security protections. URL: https://www.csoonline.com/article/573395/new-exploits-can-bypass-secure-boot-and-modern-uefi-security-protections.html.

Conti, M., Dragoni, N., Lesyk, V., 2016. A survey of man in the middle attacks. IEEE Commun. Surv. Tutorials 18 (3), 2027–2051.

Cooper, D., Polk, W., Regenscheid, A., Souppaya, M., et al., 2011. BIOS protection guidelines. NIST Spec. Publ. 800, 147.

Costin, A., 2011. Hacking MFPs. In: The 28th Chaos Communication Congress. URL: https://fahrplan.events.ccc.de/congress/2011/Fahrplan/track/Hacking/4871.en.html.

Cui, A., Costello, M., Stolfo, S.J., 2013. When firmware modifications attack: A case study of embedded exploitation. In: 20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013. The Internet Society, pp. 1–13.

Dworkin, M., 2007. Special publication 800-38d, recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac.

Embleton, S., Sparks, S., Zou, C., 2008. SMM rootkits: a new breed of OS independent malware. In: Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks. pp. 1–12.

Feng, T., Chen, T., Gong, X., 2024. Formal security analysis of ISA100. 11a standard protocol based on Colored Petri Net tool. Information 15 (2), 118.

Fuchs, A., Krauß, C., Repp, J., 2016. Advanced remote firmware upgrades using TPM 2.0. In: Proceedings of the 31th Systems Security and Privacy Protection. SEC, Springer, pp. 276–289.

Ganssle, J., 2004. The Firmware Handbook, first ed. Elsevier, Burlington, CO, USA.

Gui, Y., Siddiqui, A.S., Saqib, F., 2018. Hardware based root of trust for electronic control units. In: IEEE Region 3 Technical, Professional, and Student Conference. IEEE, pp. 1–7.

Gutmann, P., 2008. Cryptlib encryption toolkit. URL: https://www.cs.auckland.ac.nz/~pgut001/cryptlib/.

Haakegaard, R., Lang, J., 2015. The elliptic curve diffie-hellman (ECDH). URL: http://koclab.cs.ucsb.edu/teaching/ecc/project/2015Projects/Haakegaard+Lang.pdf.

Haken, I., 2015. Bypassing Local Windows Authentication to Defeat Full Disk Encryption. Black Hat Europe.

Hanna, S., Rolles, R., Molina-Markham, A., Poosankam, P., Blocki, J., Fu, K., Song, D., 2011. Take two software updates and see me in the morning: The case for software security evaluations of medical devices. In: HealthSec. pp. 6–19.

Jack, B., 2010. Jackpotting Automated Teller Machines Redux. Black Hat USA, URL: https://defcon.org/html/defcon-18/dc-18-speakers.html#Jack.

Jensen, K., Kristensen, L., 2009. CPN ML programming. In: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 43—-77. http://dx.doi.org/10.1007/b95112_3.

Jensen, K., Kristensen, L., 2015. Colored Petri Nets: a graphical language for formal modeling and validation of concurrent systems. Commun. ACM 58 (6), 61–70.

Jensen, K., Kristensen, L.M., Wells, L., 2007. Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. Int. J. Softw. Tools Technol. Transf. 9 (3–4), 213–254.

Khalid, O., Rolfes, C., Ibing, A., 2013. On implementing trusted boot for embedded systems. In: 2013 IEEE International Symposium on Hardware-Oriented Security and Trust. HOST, IEEE, pp. 75–80.

Li, Q., Chen, Y.L., 2009. Data flow diagram. In: Modeling and Analysis of Enterprise and Information Systems. Springer, pp. 85–97.

Löhr, H., Sadeghi, A.R., Winandy, M., 2010. Patterns for secure boot and secure storage in computer systems. In: 2010 International Conference on Availability, Reliability and Security. IEEE, pp. 569–573.

Maassen, A., 2009. Network bluepill-stealth router-based botnet has been ddosing dronebl for the last couple of weeks. URL: https://www.dronebl.org/blog/8.

Microsoft, 2018. Project cerberus. URL: https://github.com/Azure/Project-Cerberus.

Miller, J.F., 2013. Supply Chain Attack Framework and Attack Patterns. The MITRE Corporation, MacLean, VA.

Mitchell, C., 2005. Trusted Computing, vol. 6, Iet.

Moghimi, D., Sunar, B., Eisenbarth, T., Heninger, N., 2020. {TPM – FAIL}:{TPM} meets timing and lattice attacks. In: 29th USENIX Security Symposium. USENIX Security 20, pp. 2057–2073.

Moradi, M., Jahangir, A.H., 2024. A Petri net model for time-delay attack detection in precision time protocol-based networks. IET Cyber- Phys. Syst.: Theory Appl..

Ordinez, L., Eggly, G., Micheletto, M., Santos, R., 2020. Using UML for learning how to design and model cyber-physical systems. IEEE Rev. Iberoam. Tecnol. Del Aprendiz. 50–60.

Podder, R., Barai, R.K., 2021. Hybrid encryption algorithm for the data security of ESP32 based IoT-enabled robots. In: 2021 Innovations in Energy Management and Renewable Resources (52042). IEEE, pp. 1–5.

Podder, R., Rios, T., Ray, I., Raman, P., Righi, S., 2025. S-RFUP: Secure remote firmware update protocol. In: International Conference on Information Systems Security. Springer, pp. 42–62.

Podder, R., Sovereign, J., 2023. Project cerberus with PIT protocol. URL: https://github.com/cryptoknight13/Project-Cerberus.

Podder, R., Sovereign, J., Ray, I., Santharam, M.B., Righi, S., 2024. The PIT-cerberus framework: Preventing device tampering during transit. In: 2024 IEEE 24th International Conference on Software Quality, Reliability and Security. QRS, IEEE, pp. 584–595.

Ratzer, A., Wells, L., Lassen, H., Laursen, M., Qvortrup, J., Stissing, M., Westergaard, M., Christensen, S., Jensen, K., 2003. CPN tools for editing, simulating and analysing Coloured Petri Nets. In: The Proceeding of the Applications and Theory of Petri Nets 2003: 24th International Conference, vol. 2679, Springer, Eindhoven, The Netherlands, pp. 450–462.

Regenscheid, A., 2018. NIST SP 800-193; Platform Firmware Resiliency Guidelines. NIST, Gaithersburg, MD, USA.

Sacco, A.L., Ortega, A.A., 2009. Persistent BIOS infection. In: CanSecWest Applied Security Conference.

Schmidt, S., Tausig, M., Hudler, M., Simhandl, G., 2016. Secure firmware update over the air in the internet of things focusing on flexibility and feasibility. In: Proceeding of the Internet of Things Software Update. IoTSU, pp. 1–3.

Shostack, A., 2014. Threat Modeling: Designing for Security. John Wiley & Sons.

Vasselle, A., Maurine, P., Cozzi, M., 2019. Breaking mobile firmware encryption through near-field side-channel analysis. In: Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop. pp. 23–32.

Viega, J., Messier, M., Chandra, P., 2002. Network Security with OpenSSL: Cryptography for Secure Communications. O'Reilly Media, Inc..

Wojtczuk, R., Tereshkin, A., 2009. Attacking Intel Bios. BlackHat, Las Vegas, USA.

Zhang, J., Miao, X., 2024. Application of colored petri nets in security protocol analysis. In: International Conference on Algorithms, Software Engineering, and Network Security. pp. 676–682.

Rakesh Podder received his B.Tech. (Undergraduate) degree in Electrical Engineering from the Indian Institute of Technology (IIT), Patna, India, in 2017, and the M.E. (Masters) degree in Control System and Robotics from Jadavpur University (JU), Kolkata, India, in 2020. He is currently pursuing the Ph.D. degree in Computer Science at Colorado State University (CSU), Fort Collins, CO, USA. His research interests include cybersecurity, firmware security, AI planning, ML, and Security Analysis. He contributed to the development of secure frameworks like (PIT, S-RFUF) for Hardware Root of Trust (HRoT) and in projects such as Microsoft Project Cerberus CoRIM, googleDICE. He published papers in various international conferences and journals, and received the Best Paper Awards at the IEEE QRS 2024 and IEEE TPS 2024 Conference. He is also a student member of IEEE.

Mahmoud Abdelgawad received his B.Sc. (Undergraduate) degree in Computer Science from Benghazi University, Natural Science College, Benghazi, Libya, 1998, and his M.Sc. (Master) degree in Software Engineering from Libyan Academic of High Education, Benghazi, Libya, 2006. He is currently pursuing a Ph.D. degree in Computer Science at Colorado State University (CSU), Fort Collins, CO, USA. His research interests include Software Testing, Cybersecurity, Resiliency Analysis, and Formal Methods. He is a member of the IEEE community and a student member of ACM. He published papers in various international conferences and journals, including IEEE QRS ACM SACMAT, SpringerLink DBSec, and others.

Dr. Indrakshi Ray is a Full Professor in the Computer Science Department. She is also the Director for State Funded Cybersecurity Center for the Colorado State University System. Dr. Indrakshi Ray holds joint appointments with Electrical and Computer Engineering and Systems Engineering Department. She has been a visiting faculty at Air Force Research Laboratory, Naval Research Laboratory, and at INRIA, Rocquencourt, France. She obtained her Ph.D. in Information Technology from George Mason University. Dr. Ray's research interests include security models and protocols, formal assurance of security, and security analytics. Her current research spans energy security (smart grid, oil and natural gas), heavy vehicle security, cognitive aspects of cybersecurity, and cyber-resiliency. She has successfully graduated 19 Ph.D. and 22 M.S. students, most of whom are in the academia. She has also mentored over 22 B.S. students, many of whom are first generation and some are students with disabilities. She

has published over two hundred technical papers in refereed journals and conference proceedings with the support from agencies including Air Force Research Laboratory, Air Force Office of Scientific Research, National Institute of Health, National Institute of Standards and Technology, National Science Foundation, the United States Department of Agriculture, and industries from the US, Norway, and Japan. Dr. Ray is on the editorial board of IEEE Transactions on Services Computing, International Journal of Information Security, and Associate Editor of IEEE Security & Privacy. Dr. Ray is associated with the program committees of various conferences including ACM CCS, ACM CODASPY, ACM SACMAT, DBSec, EDBT, ESORICS, ICDE, VLDB, and WWW. She served as the program of ACM SACMAT, IFIP DBSec, and IEEE CNS. She is a senior member of the IEEE and a senior member of the ACM. She was awarded Professor Laureate from the College of Natural Sciences at Colorado State University.

Dr. Indrajit Ray is a Full Professor and Associate Chair in the Computer Science department at Colorado State University. From March 2018 until July 2021, he was a Program Director in the Secure and Trustworthy Cyberspace program at the National Science Foundation. He has more than 28 years of experience in cybersecurity, attack modeling and detection, and privacy with emphasis on security and resiliency of cyber–physical systems, security risk modeling and management, applied cryptography and protocols, human factors in security, and trust models and management. His research has been funded by the National Science Foundation, the Air Force Office of Scientific Research, the Air Force Research Laboratory, the National Institute of Health, the Department of Energy, and the Federal Aviation Administration. He has served in various program committees, review panels and journal editorial boards, published over 175 technical peer-reviewed articles, in different journals, conferences and edited volumes and has advised/ co-advised 13 Ph.D. students and more than 25 M.S. students – including several women students and students from minority and underrepresented groups – all of whom are well placed in academia and industry. He holds 4 US patents based on his research in cyber security and resiliency. Dr. Ray established the IFIP TC-11 Working Group 11.9 on Digital Forensics and served as its first Chair until 2008. Under his stewardship, IFIP WG 11.9 on Digital Forensics grew into active international community of scientists, engineers, law enforcement officials and practitioners dedicated to advancing the state of the art of research and practice in digital forensics. He is Senior Member of the IEEE, IEEE Computer Society, and the ACM.

Madhan Santharam is an accomplished technology leader with extensive experience in firmware and system software engineering. Currently serving as Director at AMI: American Megatrends International LLC in Atlanta, Georgia, Madhan has been with the organization since February 2006, playing a pivotal role in guiding firmware development and leading strategic technology initiatives. Before joining AMI in the United States, Madhan honed his expertise at AMD in Bengaluru, India, where he worked as a Firmware Development Engineer from November 2004 to February 2006. He started his professional journey at American Megatrends Technologies India (P) Ltd, serving as System Software Engineer and Module Leader from June 2000 to November 2004 in Chennai. He holds a Bachelor of Engineering degree in Electronics from the prestigious Government College of Technology (GCT) Coimbatore, completed in 2000. His leadership and extensive industry experience continue to influence advancements in firmware technologies and systems development.

Stefano Righi is an accomplished executive and expert in technology innovation with over 35 years of experience in the high-tech industry, covering all phases of firmware and software development. Currently serving as Chief Security Architect and Senior Vice President of the Global Software and Security Group at American Megatrends (AMI), Stefano is a prolific innovator with 28 patents granted by the United States Patent Office and 10 additional patents pending. He has authored six technology publications and holds a Master's degree in Electronic Engineering from the University of Bologna, Italy.