# Formal Model-Driven Discovery of Bluetooth Protocol Design Vulnerabilities

Jianliang Wu
*Purdue University*
*wu1220@purdue.edu*

Ruoyu Wu
*Purdue University*
*wu1377@purdue.edu*

Dongyan Xu
*Purdue University*
*dxu@purdue.edu*

Dave (Jing) Tian
*Purdue University*
*daveti@purdue.edu*

Antonio Bianchi
*Purdue University*
*antoniob@purdue.edu*

***Abstract*—The Bluetooth protocol suite, including Bluetooth Classic, Bluetooth Low Energy, and Bluetooth Mesh, has become the de facto standard for short-range wireless communications. While formal methods have been applied to Bluetooth security, existing efforts either focus on one configuration of a protocol or one protocol of the suite, without considering other configurations or interactions among protocols. As a result, manual analysis still dominates the state-of-the-art security research of Bluetooth specification.**

**To enable automatic Bluetooth security analysis with formal guarantees, we propose a comprehensive formal model for Bluetooth protocol suite covering both the *key sharing* phase and the *data transmission* phase, in all the three Bluetooth protocols, and detecting their design flaws automatically. Our formal model, written in ProVerif, adopts a *modular* design by abstracting each step within a protocol into an *interface* and implementing different methods in each step as *modules* to instantiate the interface, through which all possible configurations of a protocol could be examined. We further abstract different Bluetooth protocols into modules enabling the modeling of their interactions and relax the threat model to allow reasoning about semi-compromised devices. We use this model to formally verify 418 security properties and find 82 violations with attack examples capturing 5 known vulnerabilities and discovering 2 new security issues. Bluetooth SIG confirmed our independent discovery of these 2 new issues, with one issue assigned a CVE and the other issue acknowledged in a security notice. Our model provides one step towards formally verified Bluetooth security.**

## I. Introduction

The Bluetooth protocol has become the de facto standard for short-range wireless communications, powering billions of devices [1] and paving the way for its domination in the era of the Internet of Things (IoT) and 5G. The modern version of the Bluetooth protocol suite defines three wireless communication protocols: Bluetooth Classic (BC), Bluetooth Low Energy (BLE), and Bluetooth Mesh (Mesh), enabling high throughput, power saving, and peer-to-peer network respectively.

Since one design flaw in the specification can affect billions of devices, formal methods have been applied to Bluetooth security to discover design flaws. Unfortunately, existing efforts either focus on one configuration of a protocol or one protocol of the suite, and thus cannot discover vulnerabilities exploiting other configurations or in other protocols. For instance, prior works target either only one mode of Bluetooth pairing [2]–[5] or only the BLE stack [6]. Consequently, BThack [7] attacks that exploit the combination of different configurations during pairing remain unrevealed. Moreover, it is not uncommon for modern Bluetooth devices such as smartphones to support BC/BLE dual stacks, if not triple stacks (BC/BLE/Mesh). None of the existing efforts has explored the interactions between different stacks. Due to the current limitation of Bluetooth security formal analysis, manual efforts, such as BThack [7] and BadBluetooth [8], still dominate the state-of-the-art Bluetooth security research.

To enable practical and automatic Bluetooth security analysis with formal guarantees, we need to address three challenges: (*C1*): *Protocol complexity.* The complexity of the protocol makes the comprehensive modeling of the key sharing phase challenging and error-prone. For example, there are four *modes* in Secure Simple Pairing [9, p.983], such as Numeric Comparison and Just Works, and three different cases in the Passkey Entry mode [9, p.990]. (*C2*): *Co-existing protocol stacks.* Real-world devices commonly deploy dual or even triple stacks for different connectivity. The interactions among these co-existing protocol stacks will not be captured when only one protocol stack is formally verified. (*C3*): *Semi-compromised devices.* Modern Bluetooth attacks [8], [10], [11] often involve a semi-compromised device during a connection, e.g., an Android phone with a malicious app installed. Attackers do not have the root permission to manage the Bluetooth stack(s) within the device such as reading secret keys, but they could use the service provided by the Bluetooth stacks(s) to initiate or accept connections. A practical formal model needs to capture the attacker's capability precisely.

In this paper, we propose a comprehensive ProVerif [12] model for Bluetooth covering both *key sharing* and *data transmission* phases, and all three Bluetooth protocols, including BC, BLE, and Mesh. In the key sharing phase, we model the pairing protocol used by BC and BLE to agree on a shared secret key. For Mesh, we model the provisioning protocol that is used to distribute pre-generated secret keys. In the data transmission phase, we model the authentication and encryption protocols of BC, BLE, and Mesh, considering that one device might be semi-compromised.

To solve all three challenges mentioned earlier, our ProVerif model adopts a *modular* design by abstracting each step within a protocol into an *interface* and implementing different methods in each step as *modules* to instantiate the interface, through which *all* possible configurations or modes of a protocol could be examined. We further abstract different Bluetooth protocol stacks into modules enabling the modeling of their interactions, and we provide a two-layer design of the data transmission model, a stack layer and an application layer, capturing the attacker's capability accurately.

We design and implement our model using ProVerif, and formally verify Bluetooth security using 418 security properties extracted from the specification covering the two phases (key sharing and data transmission) and three different protocol stacks (BC, BLE, and Mesh). Our formal verification finds 82 violations and generates corresponding attack examples, capturing 5 known weaknesses and discovering one new vulnerability and one security issue.

The new vulnerability allows attackers to steal secret keys during Mesh provisioning. We independently discover this vulnerability, which has been confirmed by Bluetooth SIG and assigned CVE-2020-26560 [13], [14] (together with another group). The new security issue allows an attacker to launch cross-stack attacks, e.g., attacking BC through BLE. Bluetooth SIG acknowledged our findings regarding this cross-stack issue and mentioned it together with CVE-2020-15802 [15]. However, Bluetooth SIG does not consider it as a vulnerability. We also reported our findings to different Bluetooth hardware/software vendors.

In summary, this paper's contributions are as follows:

- We conduct a literature study on formal models for Bluetooth security and systematically compare different models based on coverage of phases, stacks, configurations, threat models, etc. Our study shows that no existing formal models have examined all possible configurations in SSP or Mesh, or considered a semi-compromised device during a connection.
- We design and implement a comprehensive formal model for Bluetooth security using ProVerif, covering both the *key sharing* and the *data transmission* phases, and all three Bluetooth protocols. The modular design of our model allows for enumerating every possible configuration of SSP and Mesh provisioning protocol, and capturing the attacker's ability of semi-compromising a device.
- Our formal analysis, based on the comprehensive model, has successfully reproduced 5 known vulnerabilities of BC and BLE, and discovered one new vulnerability affecting Mesh and one security issue affecting the interaction between BC and BLE. Bluetooth SIG confirmed our independent discovery of the two new issues, with one issue assigned a CVE and the other issue acknowledged in a security notice.

Our model is available publicly on GitHub [16].

## II. Background

Bluetooth is a short-range radio frequency (RF) standard for data exchange between different devices, such as smartphones, laptops, and IoT devices. Bluetooth involves three different protocols, namely Bluetooth Classic (BC), Bluetooth Low Energy (BLE), and the recently released Bluetooth Mesh (Mesh) [17]. BC and BLE devices work in a central-peripheral fashion to form a piconet [9, p.414], while Mesh allows devices to form a mesh network based on BLE advertising. For the ease of description, we also use central to refer to the device initiating a request and peripheral to refer to the device responding to a request in Mesh.

There are security mechanisms, such as encryption and authentication, in each of the Bluetooth protocols to provide confidentiality and authenticity guarantees for communication. To correctly perform encryption and authentication, key sharing is needed. When two Bluetooth devices first connect, they perform a key-sharing procedure, such as the pairing protocol for BC and BLE, to generate or distribute shared keys. During the data transmission, session keys are derived from the shared keys and are used to perform authentication and encryption.

### A. Key Sharing

Different Bluetooth protocols use different procedures to generate or distribute shared keys. BC and BLE share similar *Secure Simple Pairing* (SSP) procedure to generate the shared keys, while Mesh leverages the *provisioning* process to distribute the pre-generated keys.

**Secure Simple Pairing.** SSP introduces four modes: Just Works (JW), Out of Band (OOB), Passkey Entry (PE), and Numeric Comparison (NC). The mode is chosen based on the user I/O interfaces (e.g., screen and keyboard) of the pairing devices. Each mode has four steps: ❶ Elliptic-Curve Diffie-Hellman (ECDH) key exchange, ❷ authentication stage 1, ❸ authentication stage 2, and ❹ shared key calculation. All steps except Step ❷ are the same in the four modes. In detail, two pairing Bluetooth devices first exchange their ECC public keys and calculate the ECDH shared secret key (Step ❶). In Step ❷, each device generates a random number and exchanges it with the other device. A user action, such as pressing a button or inputting a PIN, is involved to help the devices authenticate each other in the OOB, PE, and NC mode. JW does not require such user action, and thus does not support authentication. In Step ❸, each device calculates a confirmation value and exchanges the value with the other. Then, both devices check whether the received value is correct or not. In Step ❹, both devices derive the shared key using the ECDH shared secret key and the random numbers generated in Step ❷.

**Provisioning.** Mesh mandates a two-layer encryption scheme (i.e., the network-layer encryption and application-layer encryption) and leverages a *provisioning* process to distribute the *network key*. After the distribution of the network key, the *application key*, protected by the network-layer encryption, can be distributed. During provisioning, the device that initiates provisioning and provides the keys is called the *provisioner*.

The provisioning procedure also consists of four steps, ❶ invitation, ❷ ECDH key exchange, ❸ authentication, and ❹ key distribution. In Step ❶, the provisioner discovers the Mesh device and initiates the provisioning procedure. There are two ways to perform the key exchange in Step ❷, either through an out-of-band (OOB) channel, such as Near Field Communication (NFC), or through the Bluetooth channel.

In Step ❸, there are four different authentication methods, namely Output OOB, Input OOB, Static OOB, and No OOB. In this step, the provisioner and the Mesh device generate a random number individually and exchange it. A user action, such as inputting a PIN, is involved when the Output OOB and Input OOB are used. Static OOB leverages a pre-shared secret number to help the two devices authenticate each other. The No OOB method does not require the user action or pre-shared secrets, and is therefore not authenticated.

In Step ❹, both the provisioner and the Mesh device derive a session key using the ECDH shared secret key and the random numbers generated in Step ❸. Then, the provisioner sends the network key (encrypted using the session key) to the Mesh device. This key is used for encryption at the network layer. After that, the provisioner can distribute the application key to the Mesh device. This key is used for encryption at the application layer.

Due to the different key exchange and authentication methods, there are eight different *modes* in total for provisioning. Similar to the modes in SSP, the mode is chosen based on the user I/O capabilities of the Mesh device.

TABLE I: Detailed comparison of known Bluetooth formal verification papers. **KS**: key sharing, **DT**: data transmission, **CE**: concurrent execution, **CD**: compromised device.

| Formal Verification | Stack | | | Target | | Supported Pairing Mode (BC/BLE) | | | | | CE | CD |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | BC | BLE | Mesh | KS | DT | JW | NC | PE | OOB | Mode Combination | | |
| Chang et al. [2] | ● | ● | | ● | | | ● | | | | ● | |
| Arai et al. [3] | ● | ● | | ● | | | ● | | | | ● | |
| Cremers and Jackson [5] | ● | ● | | ● | | | ● | | | | ● | |
| Sethi et al. [4] | ● | ● | | ● | | | ● | | | | | ● |
| BLE reconnection [6] | | ● | | | ● | | | | | | | |
| *Our model* | ● | ● | ● | ● | ● | ● | ● | ● | ● | ● | ◐ | ● |

◐: Support concurrent execution only in the data transmission phase.

### B. Data Transmission

Once the key sharing finishes, devices are ready to transmit data using the shared keys for authentication and encryption. If devices disconnect and reconnect, the shared keys generated during the key sharing phase are re-used. BC, BLE, and Mesh follow different procedures during data transmission.

**BC.** After the SSP, two BC devices first perform a two-way challenge-response authentication procedure to authenticate each other. During the authentication, the shared secret key is used to calculate the response. After the authentication, both BC devices derive a session key using the shared key and use the session key for encryption.

**BLE.** BLE handles data transmission differently from BC. BLE does not have the authentication procedure in data transmission. A central BLE device may perform data transmission in two different ways [6] (i.e., reactive and proactive) during the data transmission. In the reactive case, the central device sends the request to the peripheral device in plaintext and enables encryption only if an error message is received from the peripheral. In the proactive case, the central device enables encryption first and then sends the request. In both cases, to enable encryption, both the central and peripheral devices first derive a session key from the shared key.

**Mesh.** Like BLE data transmission, Mesh does not have authentication either in data transmission. After provisioning, when a central device sends data, it first performs application-layer encryption using the application key and the network-layer encryption using the network key. Then it broadcasts the encrypted data using BLE advertising.

## III. Motivation

Though Bluetooth is widely used in mobile phones, laptops, and IoT domains [1], it suffers from attacks exploiting design flaws in the specification [6]–[8], [18]–[20], such as BLESA [6] and BThack [7]. Most of the design flaws [7], [8], [18], [19] are discovered through manual analysis, which requires significant manual effort. Formal analysis, an effective *automated* approach to detect protocol design vulnerabilities, has also been applied to the Bluetooth domain.

Chang et al. [2] perform the first formal verification of SSP. Their analysis considers the Dolev–Yao [21] attack model and covers the NC mode. They model infinite SSP sessions running concurrently and identify that a device might not be correctly authenticated if the user action in one session is interpreted as the action in another session.

Arai et al. [3] model and verify an "improved" version of SSP NC mode proposed by Yeh et al. [22]. Their model also uses the Dolev–Yao threat model and supports concurrent execution of SSP. They identify that the "improved" NC mode is vulnerable to replay attacks and impersonation attacks. However, the "improved" NC mode is not adopted by the specification [9], and thus their model does not represent the majority of Bluetooth devices in the real world.

Cremers and Jackson [5] develop a new way to model Diffie-Hellman (DH) groups and thus achieve a precise and fine-grained DH key exchange modeling in SSP. Because of this precise modeling, they rediscover the invalid curve attack [23]. They also model and verify the effectiveness of different mitigations of the attack. Their model assumes the Dolev–Yao threat model, supports concurrent execution, and covers the NC mode.

Sethi et al. [4] also model and verify the NC mode of SSP. They assume that the device might be compromised, which is different from the formal analyses mentioned earlier. Based on this assumption, they propose the misbinding attack. With this attack, the user may pair with a wrong device if the device she intends to pair is compromised.

The formal analyses mentioned earlier only model and verify the key sharing phase of BC and BLE. Wu et al. [6], on the other hand, model and verify the data transmission of BLE. They identify that the reactive approach of the central BLE device in data transmission can lead to spoofing attacks. However, their model only applies to BLE.

Table I shows a detailed comparison between the existing formal analyses of Bluetooth and what we propose. As summarized in the table, prior works fail to comprehensively model Bluetooth in different ways. Prior analyses [2]–[5] of SSP only cover the NC mode without other modes, not to mention the combination of different modes and the data transmission phase. Unfortunately, attacks such as BThack [7] exploit the combination of the NC and PE modes in SSP to launch Man-in-the-Middle (MitM) attacks. Therefore, it is crucial to formally verify not only a single pairing mode but also all possible mode combinations. Besides, as Table I (the Target column) shows, prior analyses also fail to cover both the key sharing and data transmission phases. Most importantly, all prior formal analyses do not cover Mesh. These limitations call for a comprehensive model for Bluetooth protocols.

To address shortcomings of the existing formal analyses of the Bluetooth protocol, in this paper, we build a comprehensive model that covers the security-critical protocols (e.g., SSP and provisioning) during both key sharing and data transmission. Besides more realistic, the modular design of our model (see Section IV-C) allows for modeling different scenarios in a comprehensive manner, such as the different mode combinations in SSP. Finally, our model covers all Bluetooth protocols (i.e., BC, BLE, and Mesh).

As Table I (CE column) shows, our model does not support multiple sessions executing concurrently in the key sharing phase. Supporting concurrent execution for key-sharing protocols will be one of our future works, as we will discuss in Section VII. Meanwhile, our model supports concurrent execution of unlimited

sessions in the data transmission phase.

## IV. A Comprehensive Bluetooth Formal Model

In this section, we first introduce the threat model we consider, then we discuss the modeling challenges and our solutions, followed by details on our model design and implementation.

### A. Threat Model

During the key sharing between two devices, we assume the adversary has the capabilities defined in the Dolev–Yao model [21], i.e., the attacker can intercept, inject, and overhear messages transmitted in the Bluetooth physical channels between the communicating endpoints, whereas all the out-of-band channels are secure. For BC and BLE, we assume the central and peripheral devices use the latest *Secure Connections Pairing* [9, p.276] (since Bluetooth v4.2) during pairing.

In the data transmission, besides the Dolev–Yao adversary, we also consider the situation where an attacker compromises one of the endpoints by, for instance, installing a malicious app on a smartphone. In this scenario, we assume that cryptographic keys used by the Bluetooth stack (e.g., the shared secret key) cannot be obtained by an attacker. This assumption is justified by the fact that even if an attacker is able to install malicious code on an endpoint (e.g., a malicious app on an Android smartphone), they are normally not able to fully compromise the operating system (e.g., the Android OS), which manages the Bluetooth stack. However, in this scenario, the attacker can use the malicious app to communicate with already paired devices. These assumptions are in line with many previous works in the field [8], [10], [11].

Finally, we assume that the cryptographic algorithms are correctly implemented according to the specification, and the attacker is within the Bluetooth range when launching attacks.

### B. Modeling Challenges and Solutions

**C1: Protocol complexity.** Bluetooth is designed to support devices with different I/O capabilities during key sharing. The specification defines four modes in SSP and eight modes in Mesh provisioning (see Section II-A). For PE and OOB modes in SSP, each of them has three different cases. Moreover, there are 19 different scenarios in the data transmission phase (see Table IV). For instance, a laptop may connect to a speaker only for audio streaming, which needs only BC connections. A smartphone can connect to a smartwatch, which may need both BC and BLE connections. It is error-prone and challenging to comprehensively model and verify all the possible scenarios during key sharing and data transmission. Additionally, existing works only consider the NC mode of SSP in key sharing and the BLE connection in data transmission. Even worse, their NC model can hardly be reused for modeling other modes.

**Solution:** We adopt a *modular design* in our formal model to solve this challenge. Specifically, we design each sub-protocol in the SSP and provisioning protocols as an individual module. The interfaces between steps define the input and output data of a module. The modules that belong to the same step have the same interface. For example, as shown in Figure 1, the NC module in Step ❷ has the same interface as the JW module. In this case, we can replace or add modules to build the SSP and provisioning models with different modes and mode combinations. A similar modular design also applies to the modeling of the data transmission phase.

The modular design is flexible for supporting existing Bluetooth protocols. Compared with existing modeling approaches (Table I), the modules in our model are reusable. For example, the model of the NC mode of SSP in [2] can hardly be reused for modeling the PE mode, because the different steps in SSP are coupled together. In contrast, we make most of the modules in one mode reusable for the modeling of another mode. For instance, as shown in Figure 1, we reuse the modules for Step ❶, ❷, and ❹ in different SSP modes and mode combinations. The flexibility of the modular design enables us to comprehensively model all real-world cases in the key sharing phase and data transmission phase.

The modular design is extensible for future protocol modeling. If a new protocol for SSP or provisioning is introduced, most of our modules can still be used in modeling the new protocol. For example, suppose a new authentication method is introduced in Mesh provisioning. To model the new protocol, we only need to develop a new module for the new authentication method in Step ❸ (see Figure 2) while the modules in Step ❶, ❷, and ❹ can be reused.

**C2: Co-existing protocol stacks.** Different Bluetooth protocol stacks interact with each other. For instance, BC/BLE dual-stack devices support both BC and BLE connections; Mesh leverages BLE under the hood as its transport layer. To capture the interactions among different protocol stacks, we need to model all of them in a unified formal model.

**Solution:** Following the modular design of our formal model, we treat BC, BLE, and Mesh as individual modules that follow the same interface externally. The plug-and-play nature of the modules significantly alleviates the burden in modeling *all* possible co-existing stacks in the data transmission phase.

**C3: Semi-compromised devices.** In the data transmission phase, we also consider the situation where one device is compromised. A naive way to model the compromised device is to allow the attacker to access the secret keys generated in the key sharing phase by, for example, sending the keys to an open channel under the attacker's control. However, this approach does not align with the real-world scenarios (and our threat model), as we discussed in Section IV-A, where the attacker cannot obtain the keys.

**Solution:** To precisely model the attacker's capability, we propose a two-layer design of the data transmission model. Both central and peripheral devices have two layers, i.e., a stack layer and an application layer. The stack layer is responsible for authentication and encryption in data transmission, while the application layer determines what data to transmit. When the device is not compromised, the stack layer sends/receives data to/from the application layer via a secure channel. If the device is semi-compromised, besides the secure channel, the stack layer also sends/receives data to/from the application layer through an open channel. As a result, attackers cannot obtain the shared keys, but can still inject malicious inputs through the semi-compromised device.

The modular design makes it easier to model devices with different attacker's capabilities. For example, we can assess the security of Bluetooth under the semi-compromised devices attack model by replacing the stack modules with semi-compromised ones, without changing other modules.

### C. Model Design and Implementation

In this section, we elaborate on the modular design of our model and its implementation in ProVerif [12]. We first discuss how we

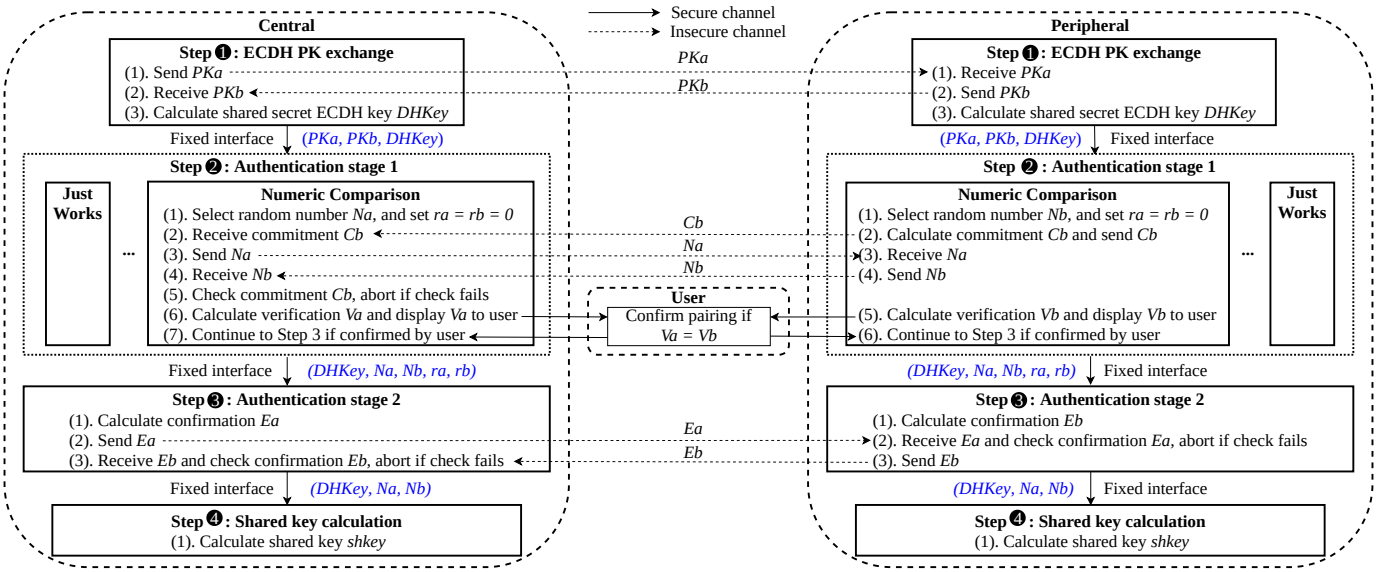


Fig. 1: Modular design of the Secure Simple Pairing (SSP) model when the Numeric Comparison (NC) mode is used. Solid rectangles indicate different modules. The text in blue shows the interfaces between steps.

```
1  //Interface between steps
2  table p1c(bt_addr, public_key, public_key, dhkey).
3  table p2c(bt_addr, random_num, random_num, random_num,
       random_num, dhkey).
4  //Central step 1
5  let step1c(pri_A: private_key) = (
6    let PKa = get_pub_key(pri_A) in ...
7    let DHKm = p256(PKb, pri_A) in
8    //Send data to Step 2
9    insert p1c(addr_B, PKa, PKb, DHKm)).
10 //Central step 2 NC mode
11 let step2cnc() = (
12   //Get data from Step 1
13   get p1c(=addr_B, PKa, PKb, DHKm) in ...
14   //Send data to Step 3
15   insert p2c(addr_B, Na, Nb, ra, rb, DHKm)).
16 //Central step 3
17 let step3c() = ( ... ).
18 //Central step 4
19 let step4c() = ( ... ).
20 //Central step 2 JW mode
21 let step2cjw() = (
22   get p1c(=addr_B, PKa, PKb, DHKm) in ...
23   insert p2c(addr_B, Na, Nb, ra, rb, DHKm)).
24 ...
25 // User action in NC mode
26 let usernc() = (
27   in(central_user_ch, va: random_num);
28   in(peripheral_user_ch, vb: random_num);
29   if va = vb then out(central_user_ch, yes);
30   out(peripheral_user_ch, yes)).
31 // Central SSP NC mode
32 step1c|step2cnc|step3c|step4c
33 // Central SSP JW mode
34 step1c|step2cjw|step3c|step4c
35 // Central SSP NC and JW mode
36 step1c|step2cnc|step2cjw|step3c|step4c
```

Listing 1: Implementation of the central device modules and the user module (NC mode) of SSP. Each module is implemented with one *process macro*. The interfaces between steps are implemented by *tables*.

design and implement the pairing model used by BC/BLE in their key sharing, followed by modeling of Mesh provisioning. At last, we detail the modeling of BC/BLE/Mesh data transmission.

*1) Modeling Secure Simple Pairing:* Figure 1 illustrates our formal model design of SSP. We define three interfaces between four steps, as the text in blue shows. We model each sub-protocol as an individual module, represented by the solid rectangles in the central and peripheral devices in the figure. The modules that belong to the same step share the same interface. Specifically, the four modules in Step ❷ have the same interfaces, indicated by `(PKa, PKb, DHKey)` and `(DHKey, Na, Nb, ra, rb)` in Figure 1. As we mentioned earlier, in some SSP modes (i.e., NC and PE), users need to perform certain actions. We also model the user action as a module (the User rectangle).

Following the Dolev–Yao attack model, we model the communication within a device (i.e., communication between steps) and the interaction between a device and the user through secure channels, represented by solid arrows. The data transmission between devices is via an open channel (dashed arrows).

We use the simplified implementation of the central device, as listed in Listing 1, to showcase how we implement the SSP model in ProVerif. We implement the interfaces using *tables* in ProVerif (Line 2 and Line 3). Each module is implemented as a *process macro* (e.g., `step1c`). The module starts by getting inputs from the pre-defined table (e.g., Line 13 and Line 22) and ends by inserting its outputs into a table (e.g., Line 9, Line 15, and Line 23). Listing 4 in the Appendix shows the full implementation of the central device modules.

We build the model of the central device in SSP NC mode by splicing the corresponding modules (Line 32 in Listing 1). With our modular design, SSP JW mode can be conveniently constructed by replacing the NC module with the JW module (Line 34 in Listing 1) at Step ❷. We can also build the SSP model with both NC and JW modes by adding the JW module (Line 36 in Listing 1). Accordingly, the whole model of SSP NC mode can be modeled as `step1c|step2cnc|step3c|step4c|step1p|step2pnc|step3p|step4p|usernc`, where the `step1p`, `step2pnc`, `step3p`, and `step4p` are the corresponding modules of the peripheral device.

*2) Modeling Mesh Provisioning:* The Mesh provisioning model follows the same design philosophy as the design of the SSP model. As illustrated in Figure 2, we define three interfaces between steps, indicated by the blue text in the figure. We model each

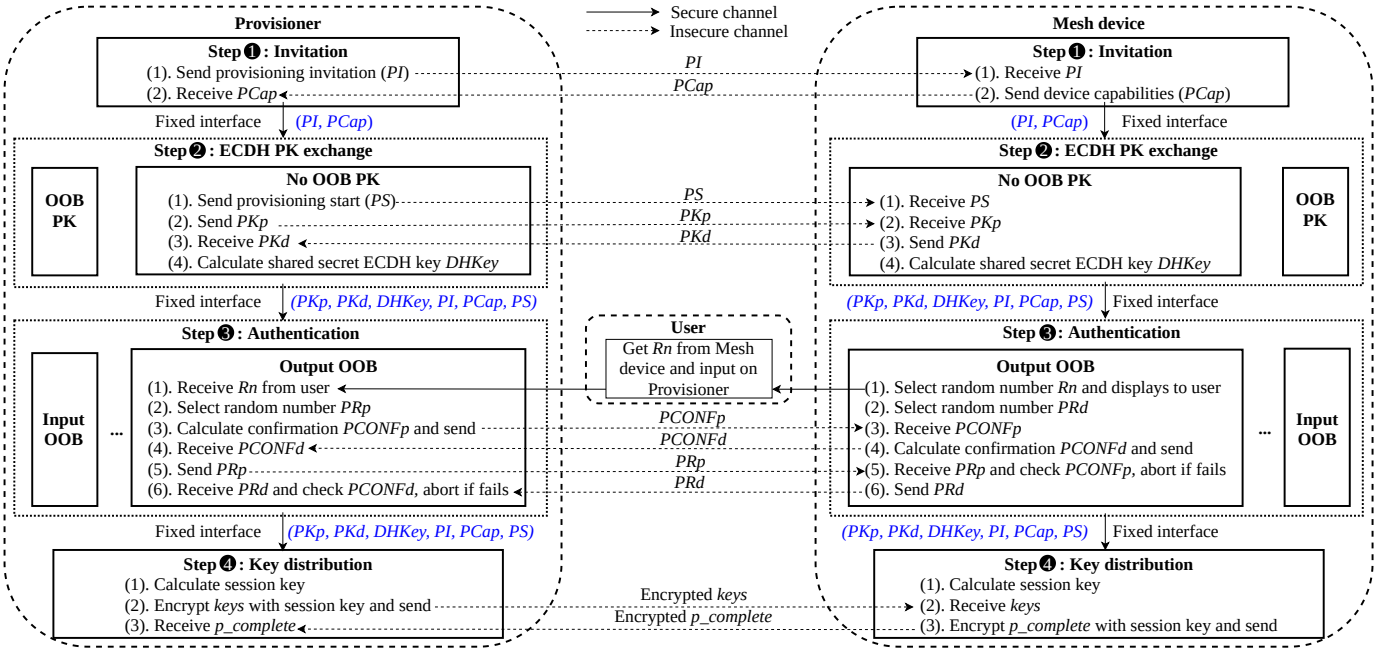


Fig. 2: Modular design of the Mesh provisioning model when No OOB public key exchange and Output OOB authentication method are used. Solid rectangles indicate the different modules, and the text in blue shows the interfaces between steps.

sub-protocol as a module, represented by the solid rectangles in the provisioner and Mesh device. The two key exchange modules in Step ❷ share the same interfaces (`(PI, PCap)` and `(PKp, PKd, DHKey, PI, PCap, PS)`). The four authentication method modules in Step ❸ also have the same interface (`(PKp, PKd, DHKey, PI, PCap, PS)`). The provisioning may also involve user interactions, which are also modeled as modules (User rectangle).

To align with the Dolev–Yao adversary capabilities, the communication within each device (i.e., communication between steps) is through secure channels (solid arrows between steps in Figure 2). The interaction between device and user is also through secure channels (solid arrows between the User and provisioner/Mesh device). The communication between the provisioner and the Mesh device is through an open channel (dashed arrows). We implement the provisioning model in a similar way to the SSP model. Using the provisioning modules, we can construct all eight (see Section II-A) possible provisioning modes infallibly.

*3) Modeling Data Transmission:* Figure 3 illustrates our two-layer design of the data transmission model, as mentioned in Section IV-B. We define a fixed interface between the stack layer and the application layer (the arrows between the stack layer and the application layer). Without losing generality, we model both central and peripheral devices using three stacks, supporting all three Bluetooth protocols (dashed rectangles in the central/peripheral device). For each Bluetooth stack, we module an application using this stack for data transmission at the application layer (dashed rounded rectangles in the central/peripheral device). We model each stack and application as a module, as the solid rectangles in central and peripheral devices indicate. To model different connection scenarios, we model the central device with three applications covering BC/BLE/Mesh, while the peripheral device may have any combination of applications.

Although the pairing is similar in BC and BLE, there is still a difference in the key derivation. If the pairing is through BC, the central/peripheral device first generates the shared key (i.e., the link key) of BC and then derives the shared key (i.e., the long-term key) of BLE. Otherwise, if the pairing is performed through BLE, the central/peripheral device generates the long-term key first and then derives the link key of BC from the long-term key of BLE. In either case, users need to perform pairing only once. Since the pairing may be done either through BC or BLE, we use two modules to capture this behavior (the key generation rounded rectangle). When the peripheral device has only a BC application, the pairing is performed through BC. Similarly, if the peripheral device has only a BLE application, the pairing is performed through BLE. The pairing is performed either via BC or BLE when the peripheral device has both BC and BLE applications. We model both cases when the peripheral device has both BC and BLE applications.

As discussed in Section IV-A, we also consider that one Bluetooth device is semi-compromised during data transmission. While this choice does not align with the assumption of Bluetooth security in the specification, it is common in the real world. For example, a smartphone may install a malicious app, or a voice assistant might be hacked [24], [25]. Without losing generality, we assume that the peripheral device might be semi-compromised while the central device is always trusted. When the device is not compromised, the communication between the stack layer and the application layer is through secure channels (the solid arrow between the stack layer and the application layer in Figure 3). If the peripheral is semi-compromised, besides the secure channel, the stack layer also communicates with the application layer through an open channel (the dashed arrow between the peripheral device's stack layer and application layer).

We first assume the key sharing phase is secure and verify the data transmission phase separately. We use predefined private keys as the generated or distributed secret keys. Considering that a composed

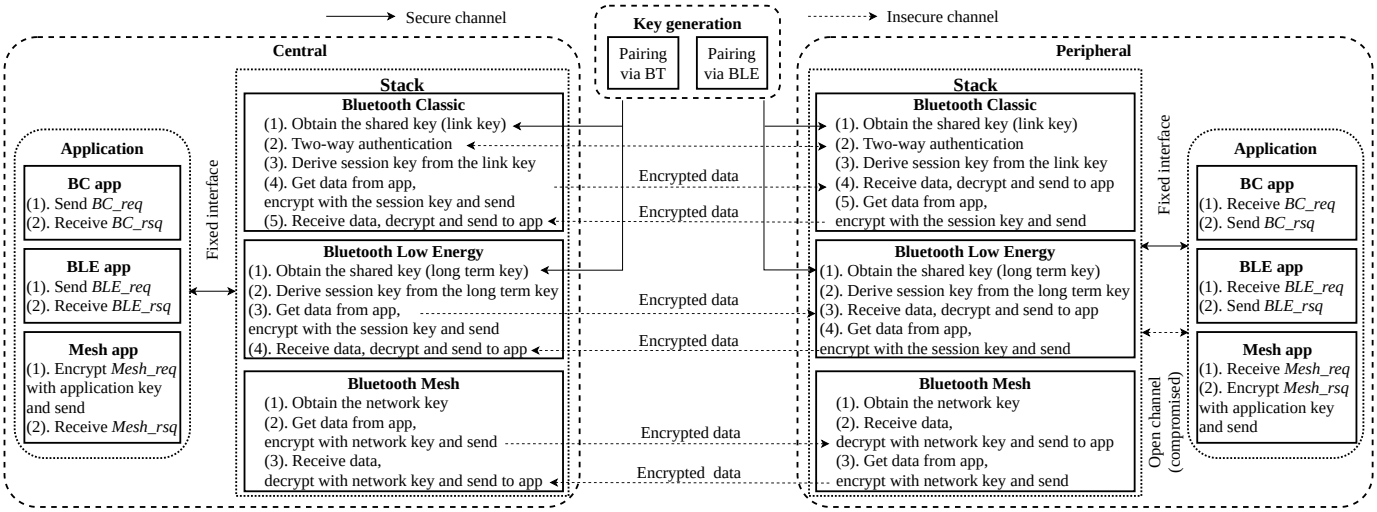


Fig. 3: Design of the data transmission model. Solid rectangles represent the modules of stacks and applications. The central and peripheral devices support all three stacks. The central device has all three applications while the peripheral device may have any combination of the applications.

protocol of two secure protocols may not be secure [26], we also verify the key sharing together with data transmission as one protocol. In this case, we splice the modules from key sharing and data transmission to generate a new protocol covering both phases. When verifying as one protocol, the secret keys used in data transmission are from the key sharing phase instead of being predefined. To better align with the real-world scenario where one device (e.g., a smartphone) can transmit data to several devices (e.g., a headset and a smartwatch) concurrently, we allow each device to have unlimited BC/BLE/Mesh sessions, which use different secret keys.

We use the implementation of the peripheral device in data transmission, as shown in Listing 5 in the Appendix, to present our implementation of the data transmission model in detail. The interfaces between the stack layer and the application layer are implemented as private channels when the peripheral is not compromised (Line 2-4). If the peripheral is semi-compromised, the interfaces are defined as open channels (Line 6 and Line 7). We implement each stack and application as a module, such as `BCP` and `BCappp` in the listing. By splicing corresponding modules, we build models for different scenarios in data transmission. For example, we splice the stack-layer modules and the BLE application module to build the model for the scenario where only BLE connections are used for data transmission (Line 67). If the peripheral device has both BC and BLE applications, we can add the BC application module (Line 69). We also implement the semi-compromised stacks as modules (e.g., `BCPC` and `BLEPC`). We replace the stack-layer modules with the compromised ones to model the semi-compromised peripheral device (Line 71).

## V. Verified Security Properties and Findings

We first describe the security goals of SSP, Mesh provisioning, and data transmission, which we extract from the specification. Then we discuss how we verify whether the protocols achieve these security goals via asserting the *authenticity* and *confidentiality* of certain messages in the protocol. After that, we present how we implement these assertions in ProVerif. At last, we discuss the result and performance of the verification. All the properties are verified on a ThinkPad X1 Yoga 3rd gen laptop with 16G RAM and an Intel i5-8350U CPU.

To ease reproducibility of our results, we created a repository [16] containing, for each table in this section (Table II, Table III, Table IV, Table V, and Table VI), a script to run our model. The same repository also contains ProVerif-generated attack traces for all the identified security violations. Additionally, for each detected violation, we provide a detailed explanation of its corresponding attack trace.

### A. Secure Simple Pairing

According to the specification [9, p.271], the NC and PE modes of SSP are authenticated and provide MitM protection. The OOB mode also provides MitM protection if the OOB channel is resistant to MitM attacks [9, p.274], which is the case in our model. To verify the authenticity of these three SSP modes, we verify the authenticity of the messages sent by the central and peripheral devices (property **A1** and **A2**). Specifically, **A1** refers to the authenticity of the central device's confirmation value (`Ea`), and **A2** represents the authenticity of the peripheral device's confirmation value (`Eb`) in Step ❸ (see Figure 1). Listing 2 in the Appendix shows how we implement the A1 and A2 assertions in ProVerif. The central device emits a `send_central` event before sending `Ea` (Line 6). The peripheral device emits a `recv_peripheral` event if the received `Ea` passes the check (Line 12). If A1 holds (the query at Line 15), we can assert that the `Ea` received by the peripheral device is always from the central device, which means the peripheral device authenticates the central device. Vice versa, the central device authenticates the peripheral device if A2 holds (the query at Line 17 in Listing 2).

Note that there might be different combinations of the three SSP modes during an SSP session. We also verify such properties in all combinations of these SSP modes by plugging in the different modules mentioned earlier. Since the JW mode is not authenticated and it is vulnerable to MitM attacks [9, p.274], A1 and A2 in SSP JW mode will be violated.

**Verification.** We first verify the security properties in each mode of SSP (Rows #1-8 in Table II). Then, we verify the properties in

TABLE II: Result and performance of the authenticity verification in different mode(s) of the SSP protocol (one session per device). **A1**: Central device authenticates peripheral device, **A2**: Peripheral device authenticates central device.

| # | Mode(s) | Property/Perf. (seconds) | | Related Attack |
|---|---|---|---|---|
| | | A1 | A2 | |
| 1 | JW | ✗/0.07 | ✗/0.06 | MitM |
| 2 | NC | ✓/0.19 | ✓/0.19 | – |
| 3 | PECoPi | ✓/0.07 | ✓/0.06 | – |
| 4 | PECiPo | ✓/0.09 | ✓/0.08 | – |
| 5 | PECiPi | ✓/0.07 | ✓/0.07 | – |
| 6 | OOBCoPi | ✓/0.04 | ✓/0.05 | – |
| 7 | OOBCiPo | ✓/0.04 | ✓/0.04 | – |
| 8 | OOBCioPio | ✓/0.06 | ✓/0.06 | – |
| 9 | NC & PECoPi | ✗/1.86 | ✗/1.73 | BThack [7] |
| 10 | NC & PECiPo | ✗/0.83 | ✗/0.72 | BThack [7] |
| 11 | NC & PECiPi | ✓/0.30 | ✓/0.28 | – |
| 12 | NC & OOBCoPi | ✓/0.25 | ✓/0.25 | – |
| 13 | NC & OOBCiPo | ✓/0.24 | ✓/0.24 | – |
| 14 | NC & OOBCioPio | ✓/0.27 | ✓/0.26 | – |
| 15 | PECoPi & OOBCoPi | ✓/0.11 | ✓/0.11 | – |
| 16 | PECoPi & OOBCiPo | ✓/0.12 | ✓/0.10 | – |
| 17 | PECoPi & OOBCioPio | ✓/0.13 | ✓/0.12 | – |
| 18 | PECiPo & OOBCoPi | ✓/0.13 | ✓/0.12 | – |
| 19 | PECiPo & OOBCiPo | ✓/0.13 | ✓/0.12 | – |
| 20 | PECiPo & OOBCioPio | ✓/0.15 | ✓/0.14 | – |
| 21 | PECiPi & OOBCoPi | ✓/0.11 | ✓/0.12 | – |
| 22 | PECiPi & OOBCiPo | ✓/0.12 | ✓/0.11 | – |
| 23 | PECiPi & OOBCioPio | ✓/0.13 | ✓/0.13 | – |
| 24 | NC & PECiPi & OOBCoPi | ✓/0.35 | ✓/0.36 | – |
| 25 | NC & PECiPi & OOBCiPo | ✓/0.36 | ✓/0.35 | – |
| 26 | NC & PECiPi & OOBCioPio | ✓/0.38 | ✓/0.37 | – |

+ PECoPi: Central outputs a random number, and the user inputs the number on the peripheral in the PE mode.
+ PECiPo: Peripheral outputs a random number, and the user inputs the number on the central in the PE mode.
+ PECiPi: User inputs the same number on both central and peripheral in the PE mode.
+ OOBCoPi: Central can output data to the OOB channel, and the peripheral can get data from the OOB channel as input in the OOB mode.
+ OOBCiPo: Peripheral can output data to the OOB channel, and the central can get data from the OOB channel as input in the OOB mode.
+ OOBCioPio: Central and peripheral can both output data to and get data from the OOB channel in the OOB mode.
+ Different rows correspond to the different usages of SSP, which are chosen based on the user interface capabilities of the central and peripheral devices.

*all* possible mode combinations (Rows #9-26 in Table II). Note that if one mode (e.g., JW) is vulnerable, the mode combination that includes this mode is also vulnerable, because the attacker can always use the vulnerable mode to perform SSP. There is no need to verify such mode combinations. In total, we verify 52 (26x2) security properties in 26 different modes and mode combinations.

Table II shows the result and performance of our verification. The security property violations in JW (#1) confirm that JW is vulnerable to MitM attacks. The generated attack trace shows that the attacker can impersonate the peripheral device and perform the peripheral device's procedures described in Section II-A to violate property A1. Similarly, the attacker can also impersonate the central device and follow the central device's procedures to violate property A2.

Our analysis also indicates that NC and PE (#2-5) are not vulnerable individually. However, in certain combinations of NC and PE (#9 and #10), A1 and A2 can be violated.

According to the ProVerif-generated attack trace, the attacker can violate A1 in #9 with the following steps. First, when the central device starts pairing, the attacker impersonates the peripheral device and uses the NC mode. Then, the attacker exchanges her public key with the central device. After that, the attacker sends a commitment value to the central device and exchanges a random number with it ((1), (2), (3), and (4) in Step ❷ in Figure 1). The central device calculates `Va` and displays it to the user. Since the central device supports both the NC and PE modes, the user may think the central device is using the PE mode in which the central displays a number. In this case, the user may directly confirm `Va`. After the user confirmation of `Va`, the attacker can exchange a confirmation with the central device and violate A1.

To violate the A2 property in #9, the attacker can conduct the following steps, given in the corresponding attack trace generated by ProVerif shown in Listing 6 in the Appendix. First, when the benign central device starts pairing, the attacker impersonates the peripheral device and uses the NC mode. Meanwhile, the attacker impersonates the central device and starts the pairing with the benign peripheral device using the PE mode. Then, the attacker exchanges her public key with both the central and peripheral devices. In the pairing with the central device using the NC mode, the attacker first follows the same procedure in SSP Step ❷ when violating the A1 property, as described previously. Since the user may think the central and peripheral devices are using the PE mode, she may input the displayed number (`Va`) on the central device into the peripheral device. As a result, the peripheral device uses `Va` as the PIN during its pairing with the attacker in the PE mode. The attacker can calculate `Va` during her pairing with the central device in the NC mode. Then, the attacker can also use `Va` as the PIN during her pairing with the peripheral device in the PE mode. Since the attacker and the peripheral device both use `Va` as the PIN, the SSP Step ❷ in the PE mode can successfully finish. At last, the attacker exchanges a confirmation with the peripheral device in Step ❸ to violate the A2 property.

Following similar procedures when violating A1 and A2 in #9, the attacker can also violate A1 and A2 in #10. These four violations correspond to the BThack attack (CVE-2020-10134) [7].

### B. Mesh Provisioning

The Mesh specification [17, p.253] states that the No OOB authentication is not authenticated, while the other three authentication methods (i.e., Output OOB, Input OOB, and Static OOB) are authenticated. Therefore, we want to verify whether the authenticated modes in provisioning guarantee authenticity. To this aim, we verify two properties related to authenticity.

Like the authenticity verification in SSP, we verify the authenticity of provisioning by asserting the authenticity of the central and peripheral devices' messages (property **A3** and **A4**). **A3** represents the authenticity of the provisioner's random number (`PRp`) in Step ❸ (see Figure 2). If A3 holds, it asserts that the `PRp` received by the Mesh device is always from the provisioner, which means the provisioner is authenticated by the Mesh device. **A4** refers to the authenticity of the Mesh device's random number (`PRd`) in Step ❸. Like A3, if A4 holds, it indicates that the Mesh device is authenticated by the provisioner. The implementation of A3 and A4 assertions is similar to the implementation in Listing 2.

Besides authenticity, we also verify the confidentiality of `keys` (property **C1**) and `p_complete` (property **C2**) messages in Step ❹ (see Figure 2). Considering the security-critical nature of `keys`, we also verify its strong secrecy [27] (**SS**) property. Listing 3 in the Appendix shows the implementation of the confidentiality assertions in ProVerif. If C1 and C2 (the queries at Line 18) hold, we assert that the attacker cannot obtain `keys` or `p_complete` in plaintext. If SS (the query at Line 19) holds,

TABLE III: Result and performance of the security property verification in the Mesh provisioning protocol (one session per device). **A3**: Provisioner authenticates the Mesh device, **A4**: Mesh device authenticates the provisioner, **C1**: Confidentiality of `keys`, **C2**: Confidentiality of `p_complete`, **SS**: Strong secrecy of `keys`.

| # | PK Ex. | Auth. Method | Property/Performance (seconds) | | | | | Related Attack |
|---|---|---|---|---|---|---|---|---|
| | | | A3 | A4 | C1 | C2 | SS | |
| 1 | OOB | O.OOB | ✗/0.17 | ✓/0.16 | ✓/0.15 | ✓/0.15 | ✓/0.54 | – |
| 2 | | I.OOB | ✗/0.11 | ✓/0.11 | ✓/0.10 | ✓/0.10 | ✓/0.29 | – |
| 3 | | S.OOB | ✗/0.10 | ✓/0.11 | ✓/0.10 | ✓/0.10 | ✓/0.25 | – |
| 4 | | N.OOB | ✗/0.11 | ✓/0.11 | ✓/0.10 | ✓/0.10 | ✓/0.26 | – |
| 5 | No OOB | O.OOB | ✗/0.41 | ✓/0.37 | ✗/0.36 | ✓/0.35 | ✗/5.28 | BlueMAN |
| 6 | | I.OOB | ✗/0.22 | ✓/0.24 | ✗/0.21 | ✓/0.20 | ✗/1.19 | BlueMAN |
| 7 | | S.OOB | ✗/0.22 | ✓/0.23 | ✗/0.22 | ✓/0.20 | ✗/0.89 | BlueMAN |
| 8 | | N.OOB | ✗/0.42 | ✗/0.40 | ✗/0.39 | ✗/0.36 | ✗/5.00 | MitM |

the attacker cannot tell the difference when `keys` changes. We verify the same properties for the No OOB authentication method.

**Verification.** We explore all eight modes of the Mesh provisioning protocol. In total, we verify 40 (8x5) security properties for the Mesh provisioning protocol.

Table III shows the result and performance of our verification of the Mesh provisioning protocol. The security violations of Row #8 in Table III confirm that No OOB authentication is not authenticated and is vulnerable to MitM attacks. However, when the OOB public key exchange is available, the Mesh device can correctly authenticate the provisioner, as Row #4 shows. This is based on the assumption that the OOB channel for public key exchange is secure, which is the case in our model. Since the Mesh device's public key is sent to the provisioner through a secure channel, the attacker cannot get the Mesh device's public key, thus cannot calculate the valid ECDH secret key and impersonate the provisioner. In this case, the provisioning protocol is immune to MitM attacks. For the same reason, when the OOB public key is available, the attacker can impersonate the Mesh device but cannot derive the session key and get the `keys` in plaintext, as indicated by Rows #1-3. The strong secrecy property of `keys` also holds when OOB public key exchange is used.

Our formal analysis also indicates that the provisioner cannot correctly authenticate the Mesh device in all eight provisioning modes (A3 column). The reason is that the attacker can use the record-and-replay approach to pass the check on the provisioner due to a design flaw in the protocol (see Section VI-A). Even worse, when the OOB public key exchange is not available, the attacker can not only bypass the check on the provisioner, but she can also get all values used to derive the session key and session nonce. As a result, the attacker can obtain the `keys` distributed to the Mesh device even when using the authenticated modes (Rows #5-7).

Based on the attack traces provided by ProVerif when detecting the violations in Rows #5-7, we develop the Bluetooth Mesh Authentication Neutralization (BlueMAN) attack. The BlueMAN attack allows the attacker to obtain the distributed keys during provisioning. We will present more details about the BlueMAN attack in Section VI-A.

### C. Data Transmission

In data transmission, the encryption is designed to provide eavesdropping protection. We verify whether the attacker can obtain the transmitted data in plaintext by verifying the confidentiality of transmitted messages. Specifically, **C3** to **C8** represent the confidentiality of `BC_req`, `BC_rsp`, `BLE_req`, `BLE_rsp`, `Mesh_req`, and `Mesh_rsp` in Figure 3.

**Verification.** We explore *all* 19 possible connection scenarios, including the ones in which one device is semi-compromised (see Section IV-C). In total, we verified 114 (19x6) security properties in the data transmission.

Table IV shows the result and performance of our verification. We note that some ProVerif-generated attack traces are the same when detecting a violation in different connection scenarios. We consider detected violations as being related to the same attack if they share the same attack trace.

When no device is compromised, the verified properties hold except C5 in Row #3. The attacker can violate this property following the steps given in the attack trace provided by ProVerif. In the reactive way of data transmission in BLE, the central device sends data (`BLE_req`) in plaintext to the peripheral device first. As a result, the attacker can obtain the data in plaintext, violating C5. This violation corresponds to the BLESA [6] attack.

Rows #2, #9, #13, and #17 show that C3 can be violated when the peripheral is semi-compromised. These violations have the same attack trace. According to the attack trace generated by ProVerif, the attacker can violate C3 with the following steps. First, when the central device sends data, the central and peripheral devices perform a two-way challenge-response authentication in which the link key (generated during pairing) is used. During the authentication, the attacker forwards the data from the central/peripheral device to the peripheral/central device. Since the central and peripheral devices share the same link key, they can successfully authenticate each other. After that, the central and peripheral devices derive the same session key and session nonce. When receiving the data (`BC_req`) from the BC app, the BC stack on the central device encrypts the data with the session key and session nonce, and sends the encrypted data to the peripheral devices. Upon receiving the encrypted data, the semi-compromised peripheral device may decrypt it with the session key and session nonce and send the decrypted data to the attacker. These violations correspond to the Mis-binding [10] and BadBluetooth [8] attacks.

Rows #5, #11, #15, and #19 show that C5 can be violated if the peripheral is semi-compromised. These violations also share the same attack trace. As the attack trace shows, the attacker can violate C5 with the following steps. When the central device sends data, it first exchanges two random numbers that are used to derive the session key and session nonce with the peripheral device. In this step, the attacker forwards the random numbers from the central/peripheral device to the peripheral/central device. As a result, the central and peripheral devices can derive the same session key and session nonce. When receiving the data (`BLE_req`) from the BLE app, the BLE stack on the central device encrypts the data with the session key and session nonce and sends the encrypted data to the peripheral devices. Upon receiving the encrypted data, the semi-compromised peripheral device may decrypt it with the session key and session nonce and send the decrypted data to the attacker. These violations correspond to the Co-located App attack [11].

Our formal analysis also reveals that in Rows #5, #11, #15, and #19, C3 can be violated when the peripheral is semi-compromised. As the attack trace shows, when the benign apps use BLE for communications, the central and peripheral devices first pair through BLE and derive the same long term key. The central and peripheral devices may also derive the link key of BC from BLE's

TABLE IV: Result and performance of the security property verification in data transmission (unlimited sessions per device). **C3**: Confidentiality of `BC_req`, **C4**: Confidentiality of `BC_rsp`, **C5**: Confidentiality of `BLE_req`, **C6**: Confidentiality of `BLE_rsp`, **C7**: Confidentiality of `Mesh_req`, **C8**: Confidentiality of `Mesh_rsp`.

| # | Connection(s) | Assumption | Verified Properties, Performance (seconds), and Related Attack (if violated) | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | BC | | BLE | | Mesh | |
| | | | C3 | C4 | C5 | C6 | C7 | C8 |
| 1 | BC-only | Not compromised | ✓/0.09 | ✓/0.10 | ✓/0.10 | ✓/0.10 | ✓/0.10 | ✓/0.10 |
| 2 | | Peripheral is semi-compromised | ✗/0.12 (BadBT, MisBd) | ✓/0.16 | ✗/0.14 (CSIA) | ✓/0.13 | ✓/0.14 | ✓/0.13 |
| 3 | BLE-only | Not compromised (reactive) | ✓/0.09 | ✓/0.10 | ✗/0.10 (BLESA [6]) | ✓/0.10 | ✓/0.10 | ✓/0.10 |
| 4 | | Not compromised (proactive) | ✓/0.09 | ✓/0.10 | ✓/0.11 | ✓/0.11 | ✓/0.13 | ✓/0.11 |
| 5 | | Peripheral is semi-compromised | ✗/0.13 (CSIA) | ✓/0.13 | ✗/0.20 (CoApp) | ✓/0.16 | ✓/0.14 | ✓/0.17 |
| 6 | Mesh-only | Not compromised | ✓/0.16 | ✓/0.18 | ✓/0.18 | ✓/0.20 | ✓/0.18 | ✓/0.18 |
| 7 | | Peripheral is semi-compromised | ✓0.26 | ✓/0.31 | ✓/0.29 | ✓/0.30 | ✓/0.30 | ✓/0.29 |
| 8 | BC & BLE (pairing via BC) | Not compromised* | ✓/0.05 | ✓/0.06 | ✓/0.06 | ✓/0.06 | ✓/0.06 | ✓/0.06 |
| 9 | | Peripheral is semi-compromised | ✗/0.07 (BadBT, MisBd) | ✓/0.08 | ✗/0.08 (CSIA) | ✓/0.08 | ✓/0.10 | ✓/0.08 |
| 10 | BC & BLE (pairing via BLE) | Not compromised* | ✓/0.05 | ✓/0.07 | ✓/0.06 | ✓/0.06 | ✓/0.06 | ✓/0.06 |
| 11 | | Peripheral is semi-compromised | ✗/0.07 (CSIA) | ✓/0.08 | ✗/0.08 (CoApp) | ✓/0.08 | ✓/0.08 | ✓/0.08 |
| 12 | BC & Mesh | Not compromised | ✓/0.18 | ✓/0.21 | ✓/0.25 | ✓/0.20 | ✓/0.19 | ✓/0.20 |
| 13 | | Peripheral is semi-compromised | ✗/0.31 (BadBT, MisBd) | ✓/0.33 | ✗/0.34 (CSIA) | ✓/0.34 | ✓/0.32 | ✓/0.32 |
| 14 | BLE & Mesh | Not compromised* | ✓/0.19 | ✓/0.21 | ✓/0.21 | ✓/0.20 | ✓/0.21 | ✓/0.20 |
| 15 | | Peripheral is semi-compromised | ✗/0.30 (CSIA) | ✓/0.33 | ✗/0.41 (CoApp) | ✓/0.43 | ✓/0.37 | ✓/0.35 |
| 16 | BC & BLE & Mesh (pairing via BC) | Not compromised* | ✓/0.18 | ✓/0.21 | ✓/0.21 | ✓/0.20 | ✓/0.21 | ✓/0.22 |
| 17 | | Peripheral is semi-compromised | ✗/0.31 (BadBT, MisBd) | ✓/0.34 | ✗/0.35 (CSIA) | ✓/0.32 | ✓/0.34 | ✓/0.34 |
| 18 | BC & BLE & Mesh (pairing via BLE) | Not compromised* | ✓/0.19 | ✓/0.24 | ✓/0.23 | ✓/0.23 | ✓/0.22 | ✓/0.26 |
| 19 | | Peripheral is semi-compromised | ✗/0.31 (CSIA) | ✓/0.40 | ✗/0.41 (CoApp) | ✓/0.39 | ✓/0.35 | ✓/0.35 |

* : BLE uses the proactive approach in data transmission.
BadBT: BadBluetooth attack [8], MisBd: Device mis-binding attack [10], CoApp: Co-located App attack [11].

long term key. Then, the attacker can follow the same procedure described when C3 is violated in Rows #2, #9, #13, and #17 to obtain the BC's data in plaintext. As a result, even though the benign apps use *BLE* for communication between the central and peripheral devices, the attacker can access the *BC* data.

Table IV also shows that C5 can be violated in Rows #2, #9, #13, and #17. As the attack trace indicates, when the benign apps use BC for communications, the central and peripheral devices first pair through BC and derive the link key. The central and peripheral devices may also derive the long term key of BLE from the BC's link key. Then, the attacker can follow the same procedure described when C5 is violated in Rows #5, #11, #15, and #19 to obtain the BC's data in plaintext. As a result, though the benign apps use *BC* for communication between the central and peripheral, the attacker can access the *BLE* data.

These eight violations (C3 in Rows #5, #11, #15, and #19, and C5 in Rows #2, #9, #13, and #17) enable the attacker to illegally access the data across stacks, i.e., BC to BLE or BLE to BC. The root cause of this issue is that the device still derives the shared key of BC (or BLE) from the shared key of BLE (or BC) even though the BC (or BLE) connections are not used (see Section VI-B). We develop the Cross Stack Illegal Access (CSIA) attack based on the attack traces provided by ProVerif when detecting these violations. We present more details about the CSIA attack in Section VI-B.

It is noteworthy that the security properties (C7 and C8) in Mesh during data transmission hold in our model even when the peripheral device is semi-compromised. While BC and BLE only require link-layer security (i.e., encryption and authentication at the link layer), the Mesh specification mandates application-layer security besides link-layer security. As a result, the attacker cannot decrypt the data (`Mesh_req`) due to the unavailability of the application key.

### *D. Key Sharing and Data Transmission*

In Section V-A, V-B, and V-C, we verify key sharing and data transmission separately. In this section, we verify key sharing and data transmission as one protocol, as mentioned earlier. We verify the same properties (i.e., **A1**, **A2**, **C3**, **C4**, **C5**, and **C6**) related to BC/BLE as we did in previous sections. For Mesh, we also verify the same set of properties (i.e., **A3**, **A4**, **C1**, **C2**, **SS**, **C7**, and **C8**) of Mesh as we did previously.

TABLE V: Result and performance of the verification of SSP and data transmission in BC and BLE as one protocol (devices are not compromised; one session per device).

| M* | Property/Performance (seconds) | | | | | |
|---|---|---|---|---|---|---|
| | A1 | A2 | C3 | C4 | C5 | C6 |
| #1 | ✗/0.97 | ✗/0.97 | ✗/0.84 | ✗/0.87 | ✗/0.86 | ✗/0.93 |
| #2 | ✓/1.08 | ✓/0.97 | ✓/0.90 | ✓/0.90 | ✓/0.95 | ✓/0.90 |
| #3 | ✓/0.46 | ✓/0.38 | ✓/0.36 | ✓/0.39 | ✓/0.36 | ✓/0.41 |
| #4 | ✓/0.43 | ✓/0.48 | ✓/0.38 | ✓/0.41 | ✓/0.38 | ✓/0.44 |
| #5 | ✓/0.39 | ✓/0.35 | ✓/0.43 | ✓/0.37 | ✓/0.40 | ✓/0.35 |
| #6 | ✓/0.27 | ✓/0.28 | ✓/0.24 | ✓/0.33 | ✓/0.25 | ✓/0.28 |
| #7 | ✓/0.27 | ✓/0.24 | ✓/0.24 | ✓/0.26 | ✓/0.25 | ✓/0.26 |
| #8 | ✓/0.40 | ✓/0.35 | ✓/0.33 | ✓/0.35 | ✓/0.35 | ✓/0.40 |
| #9 | ✗/13.09 | ✗/12.68 | ✗/12.14 | ✗/12.49 | ✗/12.96 | ✗/12.64 |
| #10 | ✗/5.46 | ✗/5.38 | ✗/5.32 | ✗/5.02 | ✗/4.91 | ✗/5.00 |
| #11 | ✓/1.38 | ✓/1.49 | ✓/1.38 | ✓/1.43 | ✓/1.29 | ✓/1.31 |
| #12 | ✓/1.39 | ✓/1.45 | ✓/1.20 | ✓/1.31 | ✓/1.30 | ✓/1.21 |
| #13 | ✓/1.37 | ✓/1.30 | ✓/1.22 | ✓/1.25 | ✓/1.22 | ✓/1.34 |
| #14 | ✓/1.36 | ✓/1.43 | ✓/1.44 | ✓/1.42 | ✓/1.45 | ✓/1.45 |
| #15 | ✓/0.79 | ✓/0.80 | ✓/0.76 | ✓/0.75 | ✓/0.80 | ✓/0.80 |
| #16 | ✓/0.74 | ✓/0.69 | ✓/0.72 | ✓/0.68 | ✓/0.70 | ✓/0.75 |
| #17 | ✓/0.97 | ✓/0.82 | ✓/0.77 | ✓/0.78 | ✓/0.81 | ✓/0.86 |
| #18 | ✓/0.83 | ✓/0.84 | ✓/0.73 | ✓/0.72 | ✓/0.74 | ✓/0.71 |
| #19 | ✓/0.74 | ✓/0.78 | ✓/0.73 | ✓/0.74 | ✓/0.81 | ✓/0.72 |
| #20 | ✓/0.81 | ✓/0.83 | ✓/0.84 | ✓/0.89 | ✓/1.08 | ✓/0.83 |
| #21 | ✓/0.70 | ✓/0.76 | ✓/0.66 | ✓/0.68 | ✓/0.69 | ✓/0.67 |
| #22 | ✓/0.71 | ✓/0.74 | ✓/0.71 | ✓/0.76 | ✓/0.68 | ✓/0.89 |
| #23 | ✓/0.85 | ✓/0.84 | ✓/0.88 | ✓/0.71 | ✓/0.74 | ✓/0.80 |
| #24 | ✓/2.08 | ✓/2.11 | ✓/1.80 | ✓/1.78 | ✓/1.96 | ✓/1.71 |
| #25 | ✓/1.94 | ✓/1.78 | ✓/1.98 | ✓/1.71 | ✓/1.69 | ✓/1.70 |
| #26 | ✓/1.94 | ✓/1.96 | ✓/1.93 | ✓/1.82 | ✓/1.99 | ✓/1.90 |

* : The corresponding mode or mode combination in Table II.

**Verification.** Similar to the verification in Section V-A, we verify the properties in *all* possible 26 different cases in BC/BLE. In total, we verify 156 (26x6) properties. Table V shows the result and performance of the verification for each property. The identified violations in Rows #1, #9, and #10 correspond to the same attacks (i.e., MitM and BThack) indicated by #1, #9, and #10 in Table II. We verify *all* 8 cases and 56 (8x7) properties for Mesh. Table VI shows the result and performance of the verification. Like the violations in Table III, the detected violations in Rows #5, #6, and #7 correspond

TABLE VI: Result and performance of the verification of the provisioning and data transmission in Mesh as one protocol (devices are not compromised; one session per device).

| M* | Property/Performance (seconds) | | | | | | |
|---|---|---|---|---|---|---|---|
| | A3 | A4 | C1 | C2 | SS | C7 | C8 |
| #1 | ✗/2.50 | ✓/2.54 | ✓/2.78 | ✓/2.90 | ✓/12.92 | ✓/2.70 | ✓/2.66 |
| #2 | ✗/1.44 | ✓/1.43 | ✓/1.49 | ✓/1.43 | ✓/7.70 | ✓/1.41 | ✓/1.42 |
| #3 | ✗/1.39 | ✓/1.49 | ✓/1.38 | ✓/1.47 | ✓/7.35 | ✓/1.37 | ✓/1.48 |
| #4 | ✗/1.40 | ✓/1.62 | ✓/1.44 | ✓/1.51 | ✓/7.81 | ✓/1.41 | ✓/1.35 |
| #5 | ✗/13.30 | ✓/13.88 | ✗/14.19 | ✓/12.92 | ✗/275.61 | ✗/13.17 | ✓/13.22 |
| #6 | ✗/3.53 | ✓/3.54 | ✗/3.43 | ✓/3.43 | ✗/43.94 | ✗/3.45 | ✓/3.38 |
| #7 | ✗/3.56 | ✓/3.38 | ✗/3.29 | ✓/3.42 | ✗/42.38 | ✗/3.29 | ✓/3.57 |
| #8 | ✗/ 287.56 | ✗/ 292.88 | ✗/ 283.27 | ✗/ 277.24 | ✗/ 15596.11 | ✗/ 279.42 | ✗/ 282.89 |

* : The corresponding mode in Table III.

to the BlueMAN attack, and the violations in Row #8 are related to MitM attacks.

## VI. Case Study

In this section, we discuss the two new attacks, Bluetooth Mesh Authentication Neutralization (BlueMAN) attack (Section VI-A) and the Cross Stack Illegal Access (CSIA) attack (Section VI-B), based on the traces generated by ProVerif during the formal verification.

### A. Bluetooth Mesh Authentication Neutralization Attack

**Assumption.** The attacker has the capabilities of the Dolev–Yao adversary and is present within the Bluetooth range during the provisioning procedure.

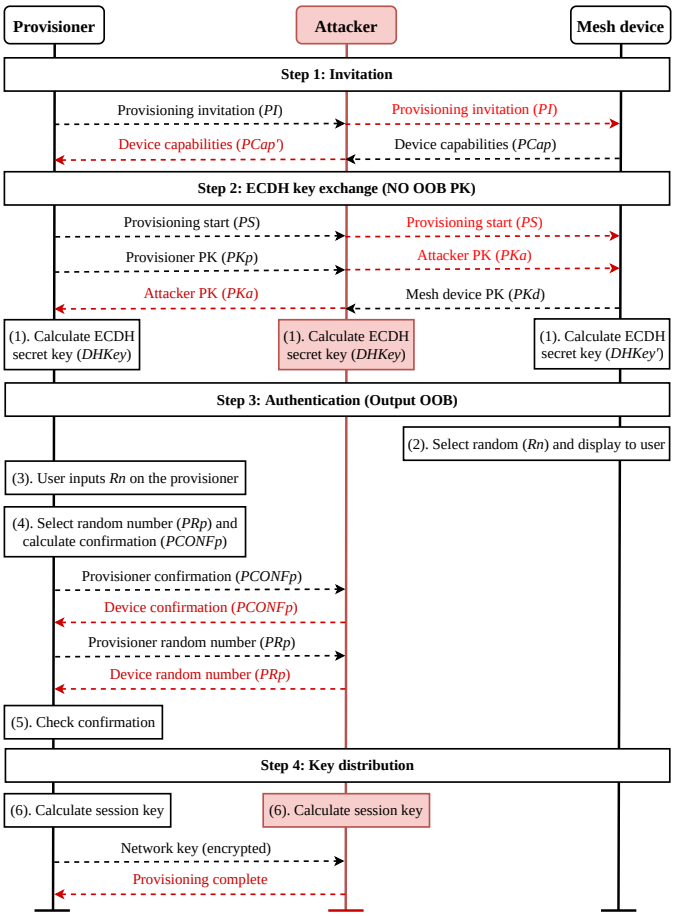


Fig. 4: Detailed procedure of the BlueMAN attack.

**Procedure.** The attacker first launches a MitM attack in the first two steps of the provisioning procedure. During the third step, the attacker can record the data from the provisioner and replay the data to the provisioner to pass the check on the provisioner. In the last step of provisioning, the attacker can derive the session key and session nonce. As a result, the attacker can decrypt the message sent by the provisioner and get the distributed keys.

Figure 4 illustrates the detailed procedure of the BlueMAN attack when the out-of-band (OOB) public key exchange is not available and the Output OOB authentication is used. The attacker acts as the MitM during the provisioning Step 1 and 2. In Step 1, the attacker receives the provisioning invitation (`PI`) from the provisioner and replays the `PI` to the Mesh device. Then, the attacker receives the device capabilities (`PCap`) from the Mesh device and sends her own device capabilities (`PCap'`) to the provisioner. During Step 2, the attacker receives the provisioner's public key (`PKp`) and sends her own public key (`PKa`) to the Mesh device. The attacker also receives the Mesh device's public key (`PKd`) and sends `PKa` to the provisioner. Accordingly, the provisioner calculates its ECDH secret key with the provisioner's private key and the attacker's public key. The attacker calculates her ECDH secret key with the attacker's private key and the provisioner's public key. Therefore, the provisioner and the attacker share the same ECDH secret key (`DHKey`).

Step 3 is the most critical step to successfully launch the attack. To better present the attack, we first describe how Step 3 of the provisioning works normally. Then, we discuss how the attacker interacts with the provisioner.

When the attacker is not present, as shown in Step ❸ Figure 2, the Mesh device selects a random number (`Rn`) and displays the number to the user. The user then inputs the number on the provisioner. After that, the provisioner selects another random number (`PRp`) and computes the confirmation value (`PCONFp`) with Equation (1), (2), and (3), where `s1()` and `k1()` functions are defined in the specification [17, p.103] based on *AES-CMAC*.

$$PCSalt = s1(PI||PCap'||PS||PKp||PKd) \quad (1)$$

$$PCKey = k1(DHKey, PCSalt, \text{“}prck\text{”}) \quad (2)$$

$$PCONFp = AES\text{-}CMAC_{PCKey}(PRp||Rn) \quad (3)$$

$$PCONFd = AES\text{-}CMAC_{PCKey}(PRd||Rn) \quad (4)$$

Then, the provisioner sends `PCONFp` to the Mesh device. After receiving `PCONFp`, the Mesh device also selects a random number (`PRd`), calculates the confirmation value (`PCONFd`) with Equation (1), (2), and (4), and sends `PCONFd` to the provisioner. Then, the provisioner sends its random number (`PRp`) to the Mesh device. Once the Mesh device receives `PRp`, it computes the provisioner's confirmation value (`PCONFp'`) with Equation (3) and checks whether `PCONFp'` is equal to `PCONFp`. If they are equal, the Mesh device sends its random number (`PRd`) to the provisioner. Once the provisioner receives `PRd`, it calculates the Mesh device's confirmation value (`PCONFd'`) with Equation (4) and checks whether `PCONFd'` is equal to `PCONFd`. If they are equal, Step ❸ is complete. Otherwise, the provisioner aborts the connection.

When the attacker is present, the provisioner computes `PCONFp` and sends it as previously described. As illustrated in Step 3 Figure 4, the attacker can record `PCONFp` and replay it to the provisioner. After that, the provisioner sends `PRp`, and the attacker also records

PRp and replays it to the provisioner. In this case, the PRd is equal to PRp, and PCONFd is equal to PCONFp. Therefore, the confirmation check on the provisioner passes (Step 3 (5)).

In Step 4, the provisioner first derives the session key and session nonce. After that, the provisioner sends the encrypted (using the session key and session nonce) secret keys. Since all the values used for the session key and session nonce derivation are known to the attacker, she can derive the session key and session nonce. Therefore, the attacker can decrypt the message to obtain the network key. The detailed attack trace provided by ProVerif is shown in Listing 7 in the Appendix. This attack is also applicable to the Input OOB and Static OOB authentication methods when the OOB public key exchange is not available.

When the OOB public key exchange is available, the attacker can still pass the confirmation check on the provisioner in Step 3 using the approach mentioned previously. However, the attacker cannot calculate the correct ECDH secret key since she is not able to replace the Mesh device's public key (PKd) with the attacker's (PKa) in Step 2. Therefore, the attacker is not able to derive the session key and nonce. Consequently, the attacker cannot decrypt the message and obtain the secret key, as shown in Table III (#1-4). In Step 3, since the Mesh device receives the provisioner's confirmation value first and then sends its own, the record-and-replay does not work for the Mesh device. Accordingly, the Mesh device can correctly authenticate the provisioner preventing the attacker from getting the message (p_complete) from the Mesh device, as Table III (#5-7) indicates.

**Impact.** We evaluate the impact of the BlueMAN attack by analyzing 25 Mesh stack implementations from different vendors (e.g., Qualcomm and Nordic Semiconductor) on different platforms (e.g., iOS, Android, WatchOS, and Bluetooth SoCs). For the stacks that we can run and perform provisioning on, we dynamically test them. Specifically, we run three stacks (i.e., IOS nRF Mesh Library [28], STSW-BNRG-Mesh-iOS [29], and meshctl in BlueZ [30]) as the provisioner and perform the provisioning with our modified Mesh device implementation. The modified Mesh device does not calculate the confirmation value. Instead, it records the confirmation value from the provisioner and replays to the provisioner. It does the same for the random number. The provisioner is affected by the BlueMAN attack if the provisioning finishes successfully.

For the stacks that we do not have a physical device to run, we either analyze their source code (if available) or analyze the binary libraries (e.g., Bluetooth SoC libraries). Some stacks (e.g., #3 and #7 in Table VIII) are runnable but cannot successfully perform the provisioning even without the attack. We also analyze either the source code or the binary libraries for those cases. Our analysis shows that out of the 25 stacks, we found 20 of them are vulnerable to the BlueMAN attack. Table VIII in the Appendix shows the details of all the analyzed stacks.

**Feasibility.** As Figure 4 shows (Step 1 and Step 2), BlueMAN requires the attacker to act as MitM to forward messages from the provisioner/device to the device/provisioner. Therefore, the most critical step to launch the attack is to make the provisioner connect to the malicious device (Attacker in Figure 4) instead of the target Mesh device in Step 1.

Different from other MitM attacks (e.g., BThack [7]), BlueMAN does not need selective jamming, which makes it more feasible, thanks to the mechanism through which the provisioner discovers the Mesh device. Since Mesh uses BLE as the underlying physical transportation, it uses the same mechanism as BLE to let the provisioner discover the Mesh device. That is, the provisioner scans for the broadcasting messages from the Mesh device to discover it. Then, to establish a connection, the provisioner sends a connection request to the Mesh device right after receiving the broadcasting message. To make the provisioner connect to the malicious device, the attacker can spoof the broadcasting message from the target Mesh device and broadcast at a much higher frequency than the target Mesh device, so that the provisioner has a higher chance to receive the broadcasting message from the malicious device and to connect to it.

**Possible fixes.** We present two possible fixes to defend against the BlueMAN attack. Since the attacker relying on recording and replaying the confirmation value and random number from the provisioner to pass the check, one straightforward and easy-to-deploy solution is that the provisioner checks whether the received confirmation value (or the random number) from the Mesh device is equal to its own confirmation value (or random number). If they are equal, the provisioner aborts the provisioning. During our analysis of the Mesh stacks, we have noticed that the Gecko_mesh v3.1 [31], Zephyr Bluetooth Mesh [32], and NimBLE [33] adopt this approach, and thus not affected by BlueMAN.

The other solution requires a minor change of Equation (4) when calculating the Mesh device's confirmation value. The BlueMAN attack can also be prevented by changing Equation (4) to Equation (5). In this case, the record-and-replay works only if Rn is equal to PRp. Considering they are 128-bit random numbers, we assume they are always different.

$$PCONFd = AES\text{-}CMAC_{PCKey}(Rn||PRd) \quad (5)$$

We also formally verify these two possible fixes and prove that both of the fixes can defend against the BlueMAN attack. All properties in Table III hold when either of the fixes is implemented in the Mesh provisioning protocol, except the ones in Row 8. In fact, Row 8 corresponds to the case in which no OOB authentication is performed on both sides. In this case, since no authentication is performed, MitM attacks are always possible.

**Responsible disclosure.** We responsibly disclosed our findings to Bluetooth SIG, and they acknowledged that we independently discovered this vulnerability with another team [14], [34]. A CVE number (CVE-2020-26560 [13]) is assigned to this vulnerability. We also reported our findings to the affected vendors in Table VIII. Tuya, Qualcomm, and Nordic Semiconductor have confirmed our findings. As of the time writing the paper, we have not received responses from other affected vendors.

### *B. Cross Stack Illegal Access Attack*

**Assumption.** The central and peripheral devices are BC/BLE dual-stack devices while communicating only via BLE. For example, a peripheral BLE app may run on a smartphone or a laptop to communicate with other central smartphones or laptops. The central and peripheral devices were paired via the Secure Connections Pairing [9, p.277] of BLE. The peripheral device is semi-compromised (see Section IV-A). The attacker aims to establish a BC connection with the victim central device and access its BC services.

**Procedure.** The Bluetooth core specification [9, p.280 and 1401] allows the link key (LK) of BC to be derived from the long-term key (LTK) of BLE, if the devices are paired through Secure Connections Pairing. Even though only the BLE connection is in use, the central and peripheral devices may still derive the LK of BC from the LTK.

Due to the symmetric nature of BC connections (both central and peripheral devices can initiate the connection), the attacker can initiate a BC connection from the semi-compromised peripheral device to the central device. If both devices derive the LK from LTK, the attacker can use the LK on the peripheral device to pass the BC authentication on the central device. Meanwhile, on modern smartphones and laptops, some BC services are enabled by default. For example, when only BLE is in use, the smartphone or laptop is also ready for communicating via BC with a headset for audio streaming or a keyboard for receiving keystrokes. As a result, the attacker can illegally access such enabled BC services on the central device.

**Impact.** We evaluate CSIA on 6 devices shown in Table VII in Appendix B. Specifically, we set up a BLE peripheral app on a Linux laptop (BC/BLE dual stack). Then, we use each of the tested devices as the central device connecting to the peripheral, during which BLE pairing is performed. Meanwhile, we also have a malicious app, which does not have root permissions (hence, it cannot access the LTK or LK), running on the Linux laptop as the semi-compromised device. The malicious app then tries to initiate connections via BC to the tested devices after the BLE pairing. Once connected, the malicious app tries to access the BC services on the tested devices, such as HID profile [35]. We consider the tested device affected by CSIA, if the malicious access to the service is successful.

Table VII in Appendix B shows the results of our evaluation. Among the six devices we test, five of them are affected by CSIA, and only Device #2 is not affected. After further investigation, we find that Device #2 is not affected due to an implementation bug [36], which leads to an incorrect LK derivation (the derived LK is reversed) and thus accidentally thwarts CSIA.

**Possible fixes.** The root cause of the CSIA attack stems from the trade-off between security and usability. Introducing the key derivation between BC and BLE allows users to pair only once when BC/BLE dual-stack devices communicate with each other. However, the key derivation fails to consider the scenario where only one type of connection is used between the dual-stack devices. Naturally, disabling the key derivation can prevent the CSIA attack, but the user needs to pair twice through BC and BLE when both connections are used. Besides, applying a fine-grained access control policy of the BC/BLE stack can also prevent the CSIA attack. For example, the central device can record the needed connection type of the peripheral device during pairing, and only allow that type of connection in the data transmission.

**Responsible disclosure.** We responsibly disclose this weakness to Bluetooth SIG, who acknowledged our findings in a security notice about CVE-2020-15802 [15], [37]. This notice, however, mentions that Bluetooth SIG does not consider this issue as a vulnerability. It states that the key derivation feature causing this issue is working as "intended". We believe that this claim stems from the fact that CSIA requires a semi-compromised device, and Bluetooth SIG does not consider this scenario in their threat model [9, p.271].

Nevertheless, we claim that the semi-compromised scenario is important in the real world, especially when involving mobile devices. In fact, mobile OSs are designed assuming that apps could potentially be malicious, and they implement mechanisms to sandbox apps' execution [38]. Under this design principle, CSIA is indeed a security issue since it allows a malicious app to access Bluetooth services that is otherwise not supposed to access.

It is worth noting that attacks requiring semi-compromised devices have been proposed in recent years, including the device mis-binding [10], BadBluetooth [8], and co-located app [11] attacks. Similar to CSIA, given their threat model, these attacks did not receive a specific CVE from Bluetooth SIG. Nevertheless, researchers have demonstrated their impact in several real-world scenarios.

## VII. Limitations and Future Work

**Simplified Diffie-Hellman modeling.** In our model, we adopt the standard modeling of the DH key exchange in ProVerif. Although DH key exchange has been verified by previous work [5], we acknowledge that such verification does not directly apply to our model. For this reason, due to the limitation of the standard DH modeling in ProVerif, our model cannot capture some properties in DH key exchange, such as associativity, and thus may miss some attacks [5] against it.

**Unlimited sessions for key-sharing protocols.** Our current model does not support concurrent execution with unlimited sessions for the key-sharing phase protocols. This limitation does not allow us to model scenarios in which one device pairs with multiple devices concurrently. For this reason, our model could miss attacks involving such concurrent pairing. As future work, we would extend our model to support unlimited sessions for the key-sharing phase protocols.

**Legacy protocols.** Some of the legacy Bluetooth protocols, such as the legacy BLE pairing and the legacy BC authentication, have been known to be broken for a long time, and the industry has been encouraged to replace them with their successor protocols. Since these legacy protocols are well studied [18], [39], we did not model and verify them. For the sake of completeness, in the future, we could easily extend our model to cover them.

**Semi-compromised device in key sharing phase.** Our current model does not consider a device compromised during the key-sharing phase. As future work, we will also consider the attack model of a semi-compromised device in the key sharing phase and explore the security implications introduced by the different capabilities of the attacker on the semi-compromised device (e.g., displaying a number to the user or inputting a number on the UI).

**Other security-related protocols in Mesh.** For Bluetooth Mesh, our current model only covers provisioning and encryption. Other security-related protocols, such as the key refresh procedure, are not included. Modeling these protocols and incorporating them with the current Mesh model will be our future work.

## VIII. Discussion

**Why using ProVerif.** First, in this paper, we focus on verifying the security properties, such as authenticity and confidentiality, in the cryptographic protocols in Bluetooth instead of other types of properties (e.g., liveness, reliability, and safety). For this reason, verification tools designed for verifying security properties, such as ProVerif, fit our needs.

Considering the scalability issue of computational models, we decided to build a symbolic model. As such, a symbolic verification tool is more suitable than a computational verification tool (e.g., CryptoVerif [40]) for our purpose.

Finally, among symbolic verification tools, ProVerif and Tamarin [41] are both well-documented and widely-used general-purpose verification tools. Each has its pros and cons, and we believe both are appropriate for our purpose. We choose ProVerif considering both its modeling features and our familiarity with this tool. Though ProVerif has some disadvantages (discussed in Section VII), it meets our overall goals for this paper.

**Completeness of the verified properties.** Given the security goals (prevention of eavesdropping and MitM attacks) of the SSP, Mesh provisioning, and encryption in the specification, we believe our security properties (i.e., confidentiality, authenticity, and strong secrecy) are complete under our attack model (see Section IV-A) for the protocols modeled. However, these properties may not be complete under a different attack model. For example, our security properties may be insufficient in a computational model, when brute-force attacks (e.g., KNOB [19]) are considered.

**CSIA and BLURtooth.** CSIA and BLURtooth [42] are caused by the same underlying issue and therefore share similarities. In fact, the underlying issue is the Cross Transport Key Derivation (CTKD) feature between BLE and BC. For this reason, both attacks require pairing via one stack (e.g., BLE) and launch attacks through another stack (e.g., BC) by abusing CTKD. However, the way of exploiting CTKD is different in these two attacks.

Specifically, in BLURtooth, an attacker can impersonate the victim peripheral and start a new pairing with the paired victim central through one stack (e.g., BLE). By doing so, the attacker forces the victim central to generate a new secret key for the victim peripheral. Due to CTKD, a new key for the victim peripheral of another stack (e.g., BC) will also be generated. This new key can overwrite the original key of this stack. In this way, by exploiting a vulnerability in the specification (affecting Bluetooth versions up to 5.1), the attacker can use a key with low entropy to overwrite a key with high entropy, or use an unauthenticated key to overwrite an authenticated key. A similar attack procedure also applies to the victim peripheral.

Unlike BLURtooth, CSIA exploits an existing pairing and the CTKD feature directly without requiring new pairings. In CSIA, we assume that the victim device is paired with a semi-compromised device. In this case, even though the app on the victim device uses only one stack (e.g., BLE) to communicate with the semi-compromised device, both the victim and the semi-compromised device generate the key of another stack (e.g., BC) due to CTKD. As such, the attacker can use this key to access the services on the victim device through the stack that is not used in benign communications. For example, a laptop may communicate with a semi-compromised phone via only BLE. With CSIA, the malicious app running on the phone can inject keystrokes to the laptop using the HID profile through BC.

As previously discussed, BLURtooth assumes that the victim device accepts new pairings, while CSIA exploits an existing pairing, thus it does not require a new pairing. Because of the new pairing requirement, BLURtooth cannot be launched in the scenario where a phone running a malicious app (semi-compromised) is paired with a laptop and does not accept new pairings, while CSIA can. On the other hand, when the victim device does accept new pairings, BLURtooth is more powerful than CSIA. In this scenario, BLURtooth can launch persistent MitM attacks against both central and peripheral victim devices, while CSIA is effective against only one device.

## IX. Related Work

**Bluetooth attacks requiring a compromised device.** Naveed et al. [10] and Xu et al. [8] discover attacks enabling the attacker to illegally access a BC device through a compromised device. The co-located app attacks [11] also require a compromised device to maliciously access a BLE device. These attacks allow malicious access within the same Bluetooth stack, either BC or BLE. The CSIA attack we presented, however, enables the attacker to maliciously access devices across the BC and BLE stacks, as shown in Table IV.

**Bluetooth attacks not requiring a compromised device.** Barnickel et al. [43], Biham et al. [23], Sun et al. [44], and Tschirschnitz et al. [7] propose MitM attacks against the SSP in BC and BLE. Lu et al. [45], [46] and Antonioli et al. [18], [19] propose attacks that can break the encryption of BC. BLESA [6] targets BLE devices and allows the attacker to send spoofed data to a BLE device. All the mentioned works do not explore the security of Bluetooth Mesh, which we found to be affected by the BlueMAN attack.

**Formal analysis of other protocols.** Researchers have also performed symbolic formal verification of other protocols, such as the TLS and Signal [47]. Bhargavan et al. formally verify TLS 1.0 [48] and 1.3 [49] using ProVerif. Kobeissi et al. [50] also use ProVerif to reason about the Signal protocol and discover new weaknesses. Girol et al. [51] conduct a comprehensive formal analysis of the Noise framework [52] using Tamarin. Their analysis uncovers previously unknown subtle differences between protocols and identifies the ones that should not be used. Three pieces of work [49]–[51] assume the compromised device attack model. However, in their attack model, the attacker can obtain the secret keys (e.g., private key) when the device is compromised, which is not the case in our attack model. Consequently, the way they model compromised devices does not apply to the model of a semi-compromised device.

## X. Conclusion

In this paper, we present a formal model of Bluetooth security-critical protocols. Our analysis adopts a modular design and a new approach to model a compromised device. The design of the model enables the first comprehensive security analysis of all the different ways the analyzed protocols can be used by different devices. Besides, to the best of our knowledge, our model is the first able to reason about Bluetooth Mesh. Using this model, we verify 418 security properties and detect 82 security violations corresponding to a new vulnerability, a new security issue, and five known attacks.

## Acknowledgments

We thank the anonymous reviewers and shepherd for their valuable comments and suggestions. This work was supported in part by the Office of Naval Research (ONR) under Grant N00014-18-1-2674. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

## References


[1] Bluetooth Special Interest Group, "2019 Bluetooth Market Update," https://www.bluetooth.com/bluetooth-resources/2019-bluetooth-market-update/, 2019, accessed: August 1, 2019.

[2] R. Chang and V. Shmatikov, "Formal Analysis of Authentication in Bluetooth Device Pairing," *Proceedings of the LICS/ICALP Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis (FCS-ARSPA)*, 2007.

[3] K. Arai and T. Kaneko, "Formal Verification of Improved Numeric Comparison Protocol for Secure Simple Pairing in Bluetooth Using ProVerif," in *Proceedings of the International Conference on Security and Management (SAM)*, 2014.

[4] M. Sethi, A. Peltonen, and T. Aura, "Misbinding attacks on secure device pairing and bootstrapping," in *Proceedings of the ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2019.

[5] C. Cremers and D. Jackson, "Prime, Order Please! revisiting Small Subgroup and Invalid Curve Attacks on Protocols Using Diffie-Hellman," in *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*, 2019.

[6] J. Wu, Y. Nan, V. Kumar, D. J. Tian, A. Bianchi, M. Payer, and D. Xu, "BLESA: Spoofing Attacks against Reconnections in Bluetooth Low Energy," in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[7] M. von Tschirschnitz, L. Peuckert, F. Franzen, and J. Grossklags, "Method Confusion Attack on Bluetooth Pairing," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2021.

[8] F. Xu, W. Diao, Z. Li, J. Chen, and K. Zhang, "BadBluetooth: Breaking Android Security Mechanisms via Malicious Bluetooth Peripherals," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[9] Bluetooth Special Interest Group, "Bluetooth Core Specifications 5.2," 2019.

[10] M. Naveed, X.-y. Zhou, S. Demetriou, X. Wang, and C. A. Gunter, "Inside Job: Understanding and Mitigating the Threat of External Device Mis-Binding on Android," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[11] P. Sivakumaran and J. Blasco, "A Study of the Feasibility of Co-located App Attacks against BLE and a Large-Scale Analysis of the Current Application-Layer Security Landscape," in *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2019.

[12] B. Blanchet, "Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif," *Foundations and Trends in Privacy and Security*, vol. 1, 2016.

[13] CVE, "CVE-2020-26560," https://cve.mitre.org/cgi-bin/cvename.cgi?name=2020-26560, 2020, accessed: April 1, 2020.

[14] Bluetooth Special Interest Group, "Bluetooth SIG Statement Regarding the 'Impersonation Attack in Bluetooth Mesh Provisioning' Vulnerability," https://www.bluetooth.com/learn-about-bluetooth/key-attributes/bluetooth-security/impersonation-mesh/, 2021, accessed: August 1, 2021.

[15] ——, "Bluetooth SIG Statement Regarding the Exploiting Cross-Transport Key Derivation in Bluetooth Classic and Bluetooth Low Energy (BLURtooth) and the Security implications of key conversion between BR/EDR and BLE Vulnerabilities," https://www.bluetooth.com/learn-about-bluetooth/bluetooth-technology/bluetooth-security/blurtooth/, 2020, accessed: November 1, 2020.

[16] "Model Implementation and Attack Trace Explanation," https://github.com/purseclab/btmodel_proverif.

[17] Bluetooth Special Interest Group, "Mesh Profile Specification 1.0.1," https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=457092, 2019, accessed: June 10, 2020.

[18] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen, "BIAS: Bluetooth Impersonation AttackS," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[19] ——, "The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR," in *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2019.

[20] ——, "Key Negotiation Downgrade Attacks on Bluetooth and Bluetooth Low Energy," *ACM Transactions on Privacy and Security*, vol. 23, no. 3, 2020.

[21] D. Dolev and A. Yao, "On the Security of Public Key Protocols," *IEEE Transactions on information theory*, vol. 29, no. 2, 1983.

[22] T.-C. Yeh, J.-R. Peng, S.-S. Wang, and J.-P. Hsu, "Securing Bluetooth Communications," *IJ Network Security*, vol. 14, no. 4, 2012.

[23] E. Biham and L. Neumann, "Breaking the Bluetooth Pairing–The Fixed Coordinate Invalid Curve Attack," in *Proceedings of the International Conference on Selected Areas in Cryptography (SAC)*, 2019.

[24] N. Roy, S. Shen, H. Hassanieh, and R. R. Choudhury, "Inaudible Voice Commands: The Long-Range Attack and Defense," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[25] N. Zhang, X. Mi, X. Feng, X. Wang, Y. Tian, and F. Qian, "Dangerous Skills: Understanding and Mitigating Security Risks of Voice-Controlled Third-Party Functions on Virtual Personal Assistant Systems," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[26] S. Ciobâca and V. Cortier, "Protocol Composition for Arbitrary Primitives," in *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*, 2010.

[27] B. Blanchet, "Automatic Proof of Strong Secrecy for Security Protocols," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2004.

[28] Nordic Semiconductor ASA, "nRF Mesh," https://apps.apple.com/us/app/nrf-mesh/id1380726771}?platform=iphone, 2020, accessed: March 10, 2021.

[29] STMICROELECTRONICS INC, "ST BLE Mesh," https://apps.apple.com/us/app/st-ble-mesh/id1348645067, 2020, accessed: March 10, 2021.

[30] BlueZ contributors, "BlueZ," http://www.bluez.org/, 2019, accessed: August 1, 2019.

[31] Silicon Labs, "Bluetooth mesh SDK 2.0.0.0 GA," https://www.silabs.com/documents/public/release-notes/bt-mesh-software-release-notes-2000.pdf, 2020, accessed: March 10, 2021.

[32] Zephyr Project Community, "Zephyr Bluetooth," https://github.com/zephyrproject-rtos/zephyr/tree/master/subsys/bluetooth/mesh, 2020, accessed: March 10, 2021.

[33] Apache, "Apache NimBLE," https://github.com/apache/mynewt-nimble, 2020, accessed: March 10, 2021.

[34] T. Claverie and J. L. Esteves, "BlueMirror: Reflections on Bluetooth Pairing and Provisioning Protocols," in *2021 IEEE Security and Privacy Workshops (SPW)*, 2021.

[35] Bluetooth Special Interest Group, "Human Interface Device Profile," https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=309012, 2020, accessed: August 1, 2020.

[36] Google, "Save reversed BR/EDR link key derived from LE LTK," https://android.googlesource.com/platform/system/bt/+/9f5d3fadbcd2447dd30a9d1df44030ad0d565b85%5E1..9f5d3fadbcd2447dd30a9d1df44030ad0d565b85/, 2020, accessed: July 26, 2021.

[37] CERT Coordination Center, "CVE-2020-15802," https://www.kb.cert.org/vuls/id/589825, 2020, accessed: November 1, 2020.

[38] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich, "The Android Platform Security Model," *ACM Transactions on Privacy and Security*, vol. 24, no. 3, 2021.

[39] M. Ryan, "Bluetooth: With Low Energy Comes Low Security," in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2013.

[40] B. Blanchet, "A Computationally Sound Mechanized Prover for Security Protocols," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2006.

[41] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The TAMARIN Prover for the Symbolic Analysis of Security Protocols," in *Computer Aided Verification (CAV)*, 2013.

[42] D. Antonioli, N. O. Tippenhauer, K. Rasmussen, and M. Payer, "BLURtooth: Exploiting Cross-Transport Key Derivation in Bluetooth Classic and Bluetooth Low Energy," arXiv, 2020.

[43] J. Barnickel, J. Wang, and U. Meyer, "Implementing an Attack on Bluetooth 2.1+ Secure Simple Pairing in Passkey Entry Mode," in *Proceedings of the IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2012.

[44] D.-Z. Sun and X.-H. Li, "Vulnerability and Enhancement on Bluetooth Pairing and Link Key Generation Scheme for Security Modes 2 and 3 ," in *International Conference on Information and Communications Security*, 2016.

[45] Y. Lu and S. Vaudenay, "Faster Correlation Attack on Bluetooth Keystream Generator E0," in *Proceedings of the International Cryptology Conference (CRYPTO)*, 2004.

[46] Y. Lu, W. Meier, and S. Vaudenay, "The Conditional Correlation Attack: A Practical Attack on Bluetooth Encryption," in *Proceedings of the International Cryptology Conference (CRYPTO)*, 2005.

[47] M. Marlinspike, "Signal on the outside, Signal on the inside," https://signal.org/blog/signal-inside-and-out/, 2016, accessed: July 26, 2021.

[48] K. Bhargavan, C. Fournet, R. Corin, and E. Zălinescu, "Verified Cryptographic Implementations for TLS," *ACM Transactions on Information and System Security*, vol. 15, no. 1, 2012.

[49] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[50] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach," in *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[51] G. Girol, L. Hirschi, R. Sasse, D. Jackson, C. Cremers, and D. Basin, "A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie-Hellman Protocols," in *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2020.
[52] T. Perrin, "The Noise Protocol Framework," https://noiseprotocol.org/noise.html, 2021, accessed: August 10, 2021.

## APPENDIX

### A. Implementations of the Model in ProVerif

Listing 2 shows the implementation of authenticity assertions in SSP. Listing 3 lists the implementation of confidentiality assertions in Mesh provisioning. Listing 4 shows the implementation of the central device modules and the user module (NC mode) of SSP. Listing 5 presents the simplified implementation of the peripheral modules and the key sharing module (pairing via BLE) in data transmission.

### B. Evaluation of BlueMAN and CSIA

TABLE VII: Result of the CSIA attack evaluation on different devices.

| # | Device | Operating System | BT Version | Vul. |
|---|---|---|---|---|
| 1 | Pixel 3 | Android 9.0 | 5.0 | Yes |
| 2 | Pixel XL | Android 10.0 | 4.2 | No |
| 3 | iPhone SE (2nd gen) | iOS 14.6 | 5.0 | Yes |
| 4 | iPad Pro (11-inch, 2nd gen) | iPadOS 14.6 | 5.0 | Yes |
| 5 | ThinkPad X1 Yoga (3rd gen) | Windows 11 (build: 22000.71) | 4.2 | Yes |
| 6 | Dell Latitude 5480 | Manjaro (kernel: 5.12) | 4.2 | Yes |

Table VIII shows the detailed result of the BlueMAN attack against different Mesh stack implementations. Table VII shows the detailed result of the CSIA attack against different devices.

### C. Attack Trace Generated by ProVerif

All the attack traces generated by ProVerif when detecting the violations and their corresponding detailed explanations can be found in our code repository [16].

Listing 6 shows the attack trace provided by ProVerif when detecting the A2 violation in Row #9 of Table II.

Listing 7 illustrates the attack trace provided by ProVerif when detecting the C1 violation in Row #5 of Table III.

```
event send_central(dhkey). event recv_central(dhkey).
event send_peripheral(dhkey).
event recv_peripheral(dhkey).
//Central step 3
let step3c() = ( ...
  event send_central(DHKm);
  out(ch, Ea); in(ch, Eb: bitstring);
  if Eb = f3(DHKm, Nb, Na, ra, iocap_B, addr_B, addr_A)
       then event recv_central(DHKm); ... ).
//Peripheral step 3
let step3p() = ( ...
  in(ch, Ea: bitstring);
  if Ea = f3(DHKs, Na, Nb, rb, iocap_A, addr_A, addr_B)
       then event recv_peripheral(DHKs);
  event send_peripheral(DHKs); out(ch, Eb); ... ).
//Authenticity query. Peripheral authenticates central.
query dhk: dhkey; event(recv_peripheral(dhk)) ==> event(
     send_central(dhk)).
//Central authenticates peripheral.
query dhk: dhkey; event(recv_central(dhk)) ==> event(
     send_peripheral(dhk)).
```

Listing 2: Authenticity assertion implementation of SSP in ProVerif.

```
free keys: bitstring [private].
free p_complete: bitstring [private].
//Provisioner step 4
let send_data_prov() = ( ...
//Derive session key and session nonce
  let sk = k1(DHKp, s1(concat(s1(concat(PI, PCap, PS,
       PKp, PKd)), PRp, PRd)), prsk) in
  let sn = k1(DHKp, s1(concat(s1(concat(PI, PCap, PS,
       PKp, PKd)), PRp, PRd)), prsn) in
  out(ch, AES_CCM(keys, sk, sn)); in(ch, r: bitstring);
  ... ).
//Mesh device step 4
let recv_data_dev() = ( ...
//Derive session key and session nonce
  let sk = k1(DHKd, s1(concat(s1(concat(PI, PCap, PS,
       PKp, PKd)), PRp, PRd)), prsk) in
  let sn = k1(DHKd, s1(concat(s1(concat(PI, PCap, PS,
       PKp, PKd)), PRp, PRd)), prsn) in
  in(ch, r: bitstring); ...
  out(ch, AES_CCM(p_complete, sk, sn)); ... ).
//Confidentiality query.
query attacker(keys). query attacker(p_complete).
noninterf keys.//Strong secrecy query.
```

Listing 3: Confidentiality assertion implementation of Mesh provisioning in ProVerif.

```
//Type definition
type random_num.
type public_key.
type private_key.
type dhkey.
type bt_addr.
//Constant values
const zero: random_num.
const btlk: bitstring.
//Interface between steps
table p1c(bt_addr, public_key, public_key, dhkey).
table p2c(bt_addr, random_num, random_num, random_num,
      random_num, dhkey).
table p3c(bt_addr, random_num, random_num, dhkey).
//Secure channel between User and central device
free central_user_ch: channel [private].
//Insecure channel between central and peripheral
free ch: channel.
//Central step 1
let step1c(pri_A: private_key) = (
  let PKa = get_pub_key(pri_A) in
  //(1) and (2): Send PKa and receive PKb.
  out(ch, PKa); in(ch, PKb: public_key);
  //(3): Calculate ECDH secret key
  let DHKm = p256(PKb, pri_A) in
  //Send data to Step 2
  insert p1c(addr_B, PKa, PKb, DHKm)).
//Central step 2 NC mode
let step2cnc() = (
  //Get data from Step 1
  get p1c(=addr_B, PKa, PKb, DHKm) in
  //(1): generate random number Na, and set ra, rb.
  new Na: random_num; let ra = zero in
  let rb = zero in
  //(2), (3), (4): receive Cb, send Na, and receive Nb
  in(ch, Cb: bitstring); out(ch, Na);
  in(ch, Nb: random_num);
  //(5): check Cb
  if Cb = f1(PKb, PKa, Nb, zero) then
  //(6): display Va to user.
  out(central_user_ch, g(PKa, PKb, Na, Nb));
  //(7): user confirms pairing
  in(central_user_ch, confirm: confirmation);
  if confirm = yes_confirm then
  //Send data to Step 3
  insert p2c(addr_B, Na, Nb, ra, rb, DHKm)).
//Central step 3
let step3c() = (
  //Get data from Step 2
  get p2c(=addr_B, Na, Nb, ra, rb, DHKm) in
  //(1): Calculate Ea
  let Ea = f3(DHKm, Na, Nb, rb, iocap_A, addr_A, addr_B
      ) in
  //(2): Send Ea, receive Eb
  out(ch, Ea); in(ch, Eb: bitstring);
  //(3): Check Eb
  if Eb = f3(DHKm, Nb, Na, ra, iocap_B, addr_B, addr_A)
       then
  //Send data to Step 4
  insert p3a(addr_B, Na, Nb, DHKm)).
//Central step 4
let step4c() = (
  //Get data from Step 3
  get p3a(=addr_B, Na, Nb, DHKm) in
  //(1): Calculate shared secret key
  let lk_key = f2(DHKm, Na, Nb, btlk, addr_A, addr_B)
      in
  insert key_table_A(addr_B, lk_key)).
//User action in NC
let usernc() = (
  //(6) in Step 2
  in(central_user_ch, va: random_num);
  in(peripheral_user_ch, vb: random_num);
  //(7) in Step 2
  if va = vb then
  out(central_user_ch, yes);
  out(peripheral_user_ch, yes)).
```

Listing 4: Implementation of the central device modules and the user module (NC mode) of SSP.

```
//Interface between stack layer and application layer
free BCappP: channel [private].
free BLEappP: channel [private].
free MeshappP: channel [private].
//Interface for compromised peripheral device
free BCappPC: channel. free BLEappPC: channel.
free MeshappPC: channel.
//Secure channel for SSP key sharing
table bc_key_s(bt_addr, key).
table le_key_s(bt_addr, key).
//BC stack
let BCP() = ( //Get link key
  get bc_key_s(=addr_A, ltk) in ...
//Receive data from central, decrypt and send to BC app
  in(ch, d: bitstring); let req = sdec(d, sk, sn) in
  out(BCappP, req); ...).
//BLE stack
let BLEP() = ( //Get long-term key (ltk)
  get le_key_s(=addr_A, ltk) in ...
//Receive
      data from central, decrypt and send to BLE app
  in(ch, d: bitstring); let req = sdec(d, sk, sn) in
  out(BLEappP, req); ...).
//Mesh stack
let MeshP() = (
//Receive
      data from central, decrypt and send to Mesh app
  in(ch, d: bitstring); let req = sdec(d, sk, sn) in
  out(MeshappP, req); ...).
//Peripheral BC app
let BCappp() = (
//Receive BC request and send BC response
  in(BCappP, req: bitstring); out(BCappP, BC_rsp)).
//Peripheral BLE app
let BLEappp() = (
//Receive BLE request and send BLE response
  in(BLEappP, req: bitstring); out(BLEappP, BLE_rsp)).
//Peripheral Mesh app
let Meshappp() = (
//Receive Mesh request and send Mesh response
  in(MeshappP, r: bitstring); out(MeshappP, Mesh_rsp)).
//Compromised BC stack
let BCPC() = (
  in(ch, d: bitstring); let req = sdec(d, sk, sn) in
  ( (out(BCappP, req); ... ) |
  //Sends/receive the data to/from the
        open channel to model the compromised BC stack
  ( out(BCappPC, req); in(BLEappPC, rsp: bitstring)));
//Compromised BLE stack
let BLEPC() = ( ...
  in(ch, d: bitstring); let req = sdec(d, sk, sn) in
  ( (out(BLEappP, req); ... ) |
  //Sends/receive the data to/from the
        open channel to model the compromised BLE stack
  ( out(BLEappPC, req); ... ));
//Compromised Mesh stack
let MeshPC() = (
  in(ch, d: bitstring); let req = sdec(d, sk, sn) in
  ( (out(MeshappP, req); ... ) |
  //Sends/receive the data to/from the
        open channel to model the compromised Mesh stack
  ( out(MeshappPC, req); ... ));
//Key sharing, pairing via BLE
let PairBLE() = (
  insert le_key_m(addr_B, ltk);
  insert le_key_s(addr_A, ltk);
  //Derive peripheral BC link key from ltk
  insert bc_key_m(addr_B, h6(h7(SALT, ltk), lebr));
  insert bc_key_s(addr_A, h6(h7(SALT, ltk), lebr))).
...
//BLE connection only
BCP|BLEP|MeshP|BLEappp|PairBLE
//Both BC and BLE connection, pairing via BLE
BCP|BLEP|MeshP|BCappp|BLEappp|PairBLE
//Compromised peripheral with only BLE connection
BCPC|BLEPC|MeshPC|BLEappp|PairBLE
```

Listing 5: Simplified implementation of the peripheral modules and the key sharing module (pairing via BLE) in data transmission.

TABLE VIII: Results of the BlueMAN attack against different Mesh stack implementations.

| # | Vendor | Stack Implementation | Platform | Analysis | Vulnerable |
|---|---|---|---|---|---|
| 1 | Nordic Semiconductor | nRF5 SDK for Mesh | nRF52840 etc. Bluetooth SoC | Source code | Yes |
| 2 | | IOS nRF Mesh Library | iOS | Dynamic | Yes |
| 3 | | Android nRF Mesh Library | Android | Binary | Yes |
| 4 | ESPRESSIF | ESP BLE MESH | ESP32 and ESP32-S Bluetooth SoC | Source code | Yes |
| 5 | STMicroelectronics | STSW-BNRG-Mesh | STM32 Nucleo etc. Bluetooth SoC | Binary | Yes |
| 6 | | STSW-BNRG-Mesh-iOS | iOS | Dynamic | Yes |
| 7 | | STSW-BNRG-Mesh-Android | Android | Binary | Yes |
| 8 | Cypress | CYW-MESH 1.0 | CYW20706 Bluetooth SoC | Binary | Yes |
| 9 | | CYW-MESH 1.0 | CYW20719 Bluetooth SoC | Binary | Yes |
| 10 | | CYW-MESH 1.0 | CYW20735 Bluetooth SoC | Binary | Yes |
| 11 | | CYW-MESH 1.0 | iOS | Binary | Yes |
| 12 | | CYW-MESH 1.0 | Android | Binary | Yes |
| 13 | | CYW-MESH 1.0 | Windows | Binary | Yes |
| 14 | | CYW-MESH 1.0 | WatchOS | Binary | Yes |
| 15 | Qualcomm | Qca4020.Or.3.0 | QCA4020 Bluetooth SoC (freertos) | Binary | Yes |
| 16 | | Qca4020.Or.3.0 | QCA4020 Bluetooth SoC (threadx) | Binary | Yes |
| 17 | Linux Foundation | BlueZ | Linux | Source code | Yes |
| 18 | | BlueZ (meshctl tool) | Linux | Dynamic | Yes |
| 19 | BlueKitchen GmbH | BlueKitchen | Embedded system | Source code | Yes |
| 20 | Tuya | Tuya IoT App SDK | Android | Binary | Yes |
| 21 | Silicon Labs | Gecko_mesh v3.1 | EFR32BG22 Series 2 Bluetooth SoC | Binary | No |
| 22 | | Gecko_mesh v3.1 | iOS | Binary | No |
| 23 | | Gecko_mesh v3.1 | Android | Binary | No |
| 24 | Linux Foundation | Zephyr OS Mesh | Zephyr OS | Source code | No |
| 25 | Apache Foundation | NimBLE | Apache Mynewt OS | Source code | No |

```
new exp_C: exponent creating exp_C_1 at {1}
new exp_P: exponent creating exp_P_1 at {2}
out(ch, ~M) with ~M = p256(gen,exp_C_1) at {6}
in(ch, a) at {7}
insert p1c(addr_B,p256(gen,exp_C_1),a,p256(a,exp_C_1)) at {9}
in(ch, gen) at {11}
out(ch, ~M_1) with ~M_1 = p256(gen,exp_P_1) at {12}
insert p1p(addr_A,gen,p256(gen,exp_P_1),p256(gen,exp_P_1)) at {14}
get p1c(addr_B,p256(gen,exp_C_1),a,p256(a,exp_C_1)) at {28}
new na: random_num creating na_8 at {15}
in(ch, HMAC_SHA256(a_1,concat(concat(a,~M),zero)))
      with HMAC_SHA256(a_1,concat(concat(a,~M),zero)) = HMAC_SHA256(a_1,concat(concat(a,p256(gen,exp_C_1)),zero)) at {18}
out(ch, ~M_2) with ~M_2 = na_8 at {19}
in(ch, a_1) at {20}
get p1p(addr_A,gen,p256(gen,exp_P_1),p256(gen,exp_P_1)) at {72}
out(central_user_data_out, SHA256(concat(concat(concat(p256(gen,exp_C_1),a),na_8),a_1))) at {24} received at {145}
out(central_user_ui, yes_confirm) at {146} received at {25}
insert p2c(addr_B,na_8,a_1,zero,zero,p256(a,exp_C_1)) at {27}
out(peripheral_user_data_in, SHA256(concat(concat(concat(p256(gen,exp_C_1),a),na_8),a_1))) at {147} received at {61}
new nb_3: random_num creating nb_8 at {62}
in(ch, HMAC_SHA256(a_2,concat(concat(gen,~M_1),SHA256(concat(concat(concat(~M,a),~M_2)
     ,a_1))))) with HMAC_SHA256(a_2,concat(concat(gen,~M_1),SHA256(concat(concat(concat(~M,a),~M_2),a_1)))) = HMAC_SHA256
      (a_2,concat(concat(gen,p256(gen,exp_P_1)),SHA256(concat(concat(concat(p256(gen,exp_C_1),a),na_8),a_1)))) at {66}
out(ch, ~M_3) with ~M_3 = HMAC_SHA256
      (nb_8,concat(concat(p256(gen,exp_P_1),gen),SHA256(concat(concat(concat(p256(gen,exp_C_1),a),na_8),a_1)))) at {67}
in(ch, a_2) at {68}
out(ch, ~M_4) with ~M_4 = nb_8 at {70}
insert p2p(addr_A,a_2,nb_8,SHA256(concat(concat(concat(p256
     (gen,exp_C_1),a),na_8),a_1)),SHA256(concat(concat(concat(p256(gen,exp_C_1),a),na_8),a_1)),p256(gen,exp_P_1)) at {71}
get p2p(addr_A,a_2,nb_8,SHA256(concat(concat(concat(p256(
     gen,exp_C_1),a),na_8),a_1)),SHA256(concat(concat(concat(p256(gen,exp_C_1),a),na_8),a_1)),p256(gen,exp_P_1)) at {100}
in(ch, HMAC_SHA256(~M_1,concat(concat(concat(concat(concat(a_2,~M_4),SHA256(concat(concat(concat(~M,a),~M_2),a_1))
     ),iocap_A),addr_A),addr_B))) with HMAC_SHA256(~M_1,concat(concat(concat(concat(concat(a_2,~M_4),SHA256(concat(concat
      (concat(~M,a),~M_2),a_1))),iocap_A),addr_A),addr_B)) = HMAC_SHA256(p256(gen,exp_P_1),concat(concat(concat(concat
      (concat(a_2,nb_8),SHA256(concat(concat(concat(p256(gen,exp_C_1),a),na_8),a_1))),iocap_A),addr_A),addr_B)) at {91}
event recv_peripheral(p256(gen,exp_P_1)) at {96} (goal)
The event recv_peripheral(p256(gen,exp_P_1)) is executed at {96}.
A trace has been found.
RESULT event(recv_peripheral(dhk)) ==> event(send_central(dhk)) is false.
--------------------------------------------------------------
Verification summary:
Query event(recv_peripheral(dhk)) ==> event(send_central(dhk)) is false.
--------------------------------------------------------------
```

Listing 6: Attack trace of the BThack attack (CVE-2020-10134) against the peripheral device.

```
new exp_P: exponent creating exp_P_1 at {1}
new exp_D: exponent creating exp_D_1 at {2}
out(ch, ~M) with ~M = PI at {5}
in(ch, a) at {6}
insert pi_table_prov(addr_dev,PI) at {7}
insert pcap_table_prov(addr_dev,a) at {8}
in(ch, a_1) at {9}
out(ch, ~M_1) with ~M_1 = PCap at {10}
insert pi_table_dev(addr_prov,a_1) at {11}
insert pcap_table_dev(addr_prov,PCap) at {12}
get pcap_table_prov(addr_dev,a) at {21}
out(ch, ~M_2) with ~M_2 = PS at {13}
out(ch, ~M_3) with ~M_3 = p256(gen,exp_P_1) at {15}
in(ch, gen) at {16}
insert pubkey_table_prov(addr_prov,p256(gen,exp_P_1)) at {18}
insert pubkey_table_prov(addr_dev,gen) at {19}
insert dhkey_table_prov(addr_prov,p256(gen,exp_P_1)) at {20}
get pi_table_dev(addr_prov,a_1) at {30}
in(ch, a_2) at {22}
in(ch, a_3) at {23}
out(ch, ~M_4) with ~M_4 = p256(gen,exp_D_1) at {25}
insert pubkey_table_dev(addr_prov,a_3) at {27}
insert pubkey_table_dev(addr_dev,p256(gen,exp_D_1)) at {28}
insert dhkey_table_dev(addr_dev,p256(a_3,exp_D_1)) at {29}
get pubkey_table_prov(addr_prov,p256(gen,exp_P_1)) at {57}
get pubkey_table_prov(addr_dev,gen) at {56}
get dhkey_table_prov(addr_prov,p256(gen,exp_P_1)) at {55}
new rand_prov: random_num creating rand_prov_2 at {31}
get pubkey_table_dev(addr_prov,a_3) at {87}
get pubkey_table_dev(addr_dev,p256(gen,exp_D_1)) at {86}
get dhkey_table_dev(addr_dev,p256(a_3,exp_D_1)) at {85}
new rand_dev_1: random_num creating rand_dev_2 at {58}
new auth_val_1: random_num creating auth_val_3 at {59}
out(dev_user_data_out, auth_val_3) at {60} received at {98}
out(prov_user_data_in, auth_val_3) at {99} received at {32}
out(ch, ~M_5) with ~M_5 = AES_CMAC(AES_CMAC(AES_CMAC(AES_CMAC(ZERO,concat(concat(
     concat(concat(PI,PCap),PS),p256(gen,exp_P_1)),gen)),p256(gen,exp_P_1)),prck),concat(rand_prov_2,auth_val_3)) at {40}
in(ch, ~M_5) with ~M_5 = AES_CMAC(AES_CMAC(AES_CMAC(AES_CMAC(ZERO,concat(concat(
     concat(concat(PI,PCap),PS),p256(gen,exp_P_1)),gen)),p256(gen,exp_P_1)),prck),concat(rand_prov_2,auth_val_3)) at {41}
event send_prov(p256(gen,exp_P_1)) at {42}
out(ch, ~M_6) with ~M_6 = rand_prov_2 at {43}
in(ch, ~M_6) with ~M_6 = rand_prov_2 at {44}
event recv_prov(p256(gen,exp_P_1)) at {46}
insert key_table_prov(addr_dev,AES_CMAC(AES_CMAC(AES_CMAC(ZERO,concat(concat(AES_CMAC(ZERO,concat(concat
     (concat(concat(PI,PCap),PS),p256(gen,exp_P_1)),gen)),rand_prov_2),rand_prov_2)),p256(gen,exp_P_1)),prsk)) at {51}
insert nonce_table_prov(addr_dev,AES_CMAC(AES_CMAC(AES_CMAC(ZERO,concat(concat(AES_CMAC(ZERO,concat(concat
     (concat(concat(PI,PCap),PS),p256(gen,exp_P_1)),gen)),rand_prov_2),rand_prov_2)),p256(gen,exp_P_1)),prsn)) at {54}
get key_table_prov(addr_dev,AES_CMAC(AES_CMAC(AES_CMAC(ZERO,concat(concat(AES_CMAC(ZERO,concat(concat
     (concat(concat(PI,PCap),PS),p256(gen,exp_P_1)),gen)),rand_prov_2),rand_prov_2)),p256(gen,exp_P_1)),prsk)) at {92}
get nonce_table_prov(addr_dev,AES_CMAC(AES_CMAC(AES_CMAC(ZERO,concat(concat(AES_CMAC(ZERO,concat(concat
     (concat(concat(PI,PCap),PS),p256(gen,exp_P_1)),gen)),rand_prov_2),rand_prov_2)),p256(gen,exp_P_1)),prsn)) at {91}
out(ch, ~M_7) with ~M_7 = AES_CCM(keys,AES_CMAC(AES_CMAC(AES_CMAC
     (ZERO,concat(concat(AES_CMAC(ZERO,concat(concat(concat(concat(concat(PI,PCap),PS),p256(gen,exp_P_1)),gen)),rand_prov_2
     ),rand_prov_2)),p256(gen,exp_P_1)),prsk),AES_CMAC(AES_CMAC(AES_CMAC(ZERO,concat(concat(AES_CMAC(ZERO,concat(concat
     (concat(concat(PI,PCap),PS),p256(gen,exp_P_1)),gen)),rand_prov_2),rand_prov_2)),p256(gen,exp_P_1)),prsn)) at {88}
The attacker has the message sdec(~M_7,AES_CMAC(AES_CMAC(AES_CMAC(ZERO,concat(concat(AES_CMAC(ZERO
     ,concat(concat(concat(concat(~M,~M_1),~M_2),~M_3),gen)),~M_6),~M_6)),~M_3),prsk),AES_CMAC(AES_CMAC(AES_CMAC(ZERO,
     concat(concat(AES_CMAC(ZERO,concat(concat(concat(concat(~M,~M_1),~M_2),~M_3),gen)),~M_6),~M_6)),~M_3),prsn)) = keys.
A trace has been found.
RESULT not attacker(keys[]) is false.
------------------------------------------------------------
Verification summary:
Query not attacker(keys[]) is false.
------------------------------------------------------------
```

Listing 7: Attack trace of the BlueMAN attack (CVE-2020-26560) when the OOB public key exchange is not available and the Output OOB authentication method is used.