POLITECNICO DI TORINO
Repository ISTITUZIONALE

FDO Protocol: A Possible Solution to Protect the Ownership Voucher Against Untrusted Supply Chain

Original
FDO Protocol: A Possible Solution to Protect the Ownership Voucher Against Untrusted Supply Chain / Bussa, S., Sisto, R., Valenza, F.. - In: IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING. - ISSN 1545-5971. - STAMPA. - 23:4(2026), pp. 8403-8417. [10.1109/tdsc.2026.3683107]

Availability:
This version is available at: 11583/3017894 since: 2026-11-13T14:29:21Z

Publisher:
Institute of Electrical and Electronics Engineers

Published
DOI:10.1109/tdsc.2026.3683107

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

19 November 2026

FDO Protocol: A Possible Solution to Protect the Ownership Voucher Against Untrusted Supply Chain

Simone Bussa, Riccardo Sisto, *Member, IEEE*, and Fulvio Valenza, *Member, IEEE*

***Abstract*—The FDO protocol is the leading candidate to become the reference standard for automatic device onboarding. However, as documented in the literature, it may suffer from a security issue exploiting its Ownership Voucher when the supply chain of the device cannot be considered trusted. In a previous work, we showed a possible attack of this type, found by performing a formal analysis on a symbolic model of the protocol. The first contribution of this paper is to present the analysis done in the previous work in a more comprehensive way, to extend it, and to show that the attack found is also possible on a real implementation of the protocol. We then analyse some possible solutions proposed in the literature as countermeasures to try to mitigate the attack. After demonstrating that they may not completely solve the mentioned issue, we propose our own solution, which could be taken into consideration by the FIDO Working Group to further improve the protocol specification. Again, as a demonstration of the logical correctness of our solution, we formally analyse its symbolic model.**

***Index Terms*—Automatic device onboarding, FIDO FDO, formal verification, IoT devices, security analysis.**

I. Introduction

THE number of Internet of Things devices is growing exponentially and is expected to reach 20 billion connected devices by the end of 2025 [1]. This impressive IoT growth is facilitated by the development of important technologies, such as 5G and cloud computing. Rising connectivity in terms of speed and coverage and the computational capabilities offered by the cloud are contributing to the emergence of new IoT use cases and solutions. In this context, the greater volume and complexity of data being collected, transmitted, and stored by different devices imply the need for greater security and privacy, as witnessed by numerous documents [2], [3].

In this paper, we focus on a significantly critical phase in the life cycle of an IoT device, namely the Onboarding or Provisioning. By definition, IoT Device Onboarding is the process by which a newly powered-on device establishes a first reliable connection with the platform on which it has to operate. The platform can be either an on-premise site or a cloud platform, typically used to collect and analyze data from the device. During the Onboarding phase, the device and the platform must connect, authenticate each other, and exchange cryptographic material to encrypt future communications.

Received 27 May 2024; revised 23 February 2026; accepted 1 April 2026. Date of publication 13 April 2026; date of current version 9 July 2026. *(Corresponding author: Fulvio Valenza.)*

The authors are with the Politecnico di Torino, Dip. Automatica e Informatica, 10129 Torino, Italy (e-mail: simone.bussa@polito.it; riccardo.sisto@polito.it; fulvio.valenza@polito.it).

Digital Object Identifier 10.1109/TDSC.2026.3683107

Most onboarding processes nowadays are still performed manually, with a skilled operator personally entering the credentials into the device, which are then used by the device to authenticate to the platform. This type of operation is slow, expensive, and requires trust in the operator who must know the credentials to enter in clear. Moreover, it is a non-scalable process, which is suitable for configuring a few devices, but becomes costly when the number of devices to be configured is high, as is often the case in today's scenarios. On the other hand, as an alternative, automatic solutions have numerous challenges to face. The main ones are the vast heterogeneity of device hardware and software and the huge number of platform types on which the device can onboard. Zero-touch processes that exist nowadays tend to work exclusively with specific cloud platforms or silicon device vendors [4]. As a result, a company dealing with several devices from different vendors, which need to connect them to various platforms, must interface with many proprietary protocols, inheriting most of the disadvantages typical of the manual approach (e.g., time and need for skilled staff). Other existing solutions, on the other hand, require the final owner of the device to be known at manufacturing time, so that the device can be preconfigured during silicon fabrication with the credentials it will use to authenticate with the prefixed platform. All such solutions severely limit flexibility.

As it stands today, there is not yet an accepted standard shared by all IoT device vendors and interoperable across various platforms. Starting in 2020, the National Institute of Standards and Technology (NIST) has launched a project to try to address this gap [5]. To date, the project is still a work in progress. In the same year, the FIDO Alliance, supported by most of the global tech leaders (e.g., Intel, Amazon, Google, Microsoft, etc., just to name the biggest ones), created a working group to develop a standard solution for automatic device onboarding. The group chose to base the new protocol, called FIDO Device Onboard (FDO), on an existing one, namely the Intel Secure Device Onboard (SDO) [6]. FIDO released the first specification of the new protocol in March 2021 (v1.0, [7]), followed by an updated version in April 2022, which is a Proposed Standard Specification (v1.1, [8]). FDO overcomes most of the limitations of manual approaches and proprietary protocols: it is a zero-touch vendor-independent solution that can be adopted by any platform (cloud, on-premise, etc.), and that aims to reconcile all proprietary protocols under a single standard. The goal is to have a semi-skilled operator bring the device to the installation point and simply connect it to the Internet and turn it on. The rest of the provisioning is automatic, with the device able to find the

© 2026 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

platform and perform the connection without other manual intervention. FDO also implements late binding, which allows the binding between the device and the platform to occur in a later stage, without the need to configure the device with information about its final destination already during manufacturing. To reach this goal, a special certificate structure, called Ownership Voucher (OV), is used to keep track of all ownership changes that occur to a device during its life cycle. Before reaching its final platform, in fact, a device can be purchased and sold by a number of intermediate owners (e.g., manufacturer, warehouses, retailers, etc.), that form the so-called supply chain of the device. The OV allows to flexibly add new entries in the supply chain, thus permitting the device to change owner an unlimited number of times. At the end of the path, whoever proves to be the rightful owner of the OV has the right to onboard the device. This important feature allows us to overcome the limitation of requiring a device to be pre-configured with information about its final owner during manufacturing.

FIDO Device Onboard is the leading candidate to become the reference standard for automatic device onboarding. For this reason, it is really crucial to analyse its security. In a preliminary conference paper [9], we used Proverif to formally analyse various security properties of FDO under different scenarios. The results confirmed the guess, already mentioned by the FIDO Alliance [10], that the FDO specification [8] may suffer from security issues when the supply chain of the device cannot be assumed as trusted, and in that paper we showed a specific possible attack. Stemming from these preliminary results, a first contribution of this paper is to extend the analyses made in our previous work with a more in-depth investigation of the possible consequences that could result from the discovered security issue. Moreover, since the formal analysis in [9] was performed on a symbolic model of the protocol, which neglects some low-level aspects and may be subject to false positives (i.e., it can return false attacks that do not really exist on the actual protocol), we report the tests we made on a real implementation of the FDO to confirm that the attacks found can really be performed.

A second contribution is a critical analysis of some possible countermeasures proposed by the FIDO Alliance, in the app note [10], which can be used to build a solution that mitigates the security issue. We demonstrate that, even with the suggested countermeasures in place, an attack against the FDO protocol remains possible. At this point, we propose a solution that can definitely solve all the related problems, as a third contribution. To provide security assurance about the proposed solution, we use formal verification again and the same verification scenarios under which we found the attack against FDO in our initial work [9].

The remainder of this paper is structured as follows. Section II analyses related work. Section III describes the FDO protocol specification [8]. Section IV summarises the main threats in the FDO threat model. Section V presents the security issue related to an untrusted OV and analyses all its consequences in detail. In addition, it describes the attack tests conducted against the protocol implementation. Section VI discusses the countermeasures suggested by FIDO to solve the problem and demonstrates that an attack is still possible. Section VII presents our proposed solution. Finally, Section VIII describes the formal analysis of the protocol and our proposed solution, and Sections IX and X draw the conclusions.

TABLE I
FDO SECURITY REQUIREMENTS

Requirements and threats	
Security goal	**Way to achieve it**
Secure communication	Strong authentication of all actors. Whenever possible, use http over tls
Secure storage of credentials	Sensitive data must be stored in a secure environment (e.g., SE, TPM)
Software integrity	Mechanisms to verify the integrity of the device (e.g., attestation protocols)
Confidential data	Sensitive data must be encrypted during communications
Threat	**Protection mechanism**
Replay of data	Nonces to ensure freshness of messages
Injection of data	Digital signatures to ensure data integrity
Deletion of data	-
MitM	Authentication of the parties
Physical attacks	Critical operations performed in a secure environment
Disclosure of sensitive data	Encryption of confidential messages

II. RELATED WORK

Several studies have been published over the years analysing the security of IoT devices, e.g., [2], [11], [12]. Despite listing many of the key security and privacy properties related to IoT devices, these papers only make general considerations, without focusing on the onboarding phase, nor even mentioning the FDO protocol. Being FDO a very recent specification, we found few analyses in the literature about its security.

A first official analysis of the FDO protocol was made by FIDO in the document [13]. Here, the Alliance presented a first threat model describing the security and privacy requirements of the protocol. Table I summarises the main security goals to be satisfied along with the way to achieve them, extracted from [13]. It also includes a series of well-known threats that may target the protocol, and for which the protocol adopts specific protection mechanisms to fulfil the security properties. In [14], instead, the FIDO Alliance published a set of app notes to be considered when developing an implementation of the FDO. These include guidance on security aspects and considerations about potential security and privacy issues that may affect the protocol specification. All these elements are part of a plan for an intensive certification program that FIDO has been developing over the past few years. Vendors must follow it to get a certified FIDO. At this stage, only the most basic security certification has been defined to prevent widespread attacks. Nevertheless, this level will be gradually increased in the near future by introducing more stringent guidelines and enhanced security assurance, relying on feedback, analysis, and continuous validation that will be done on the protocol.

As mentioned in the Introduction, in [9] we performed a first formal analysis of the FDO protocol. The FIDO Alliance itself confirmed that it was the first formal security verification work

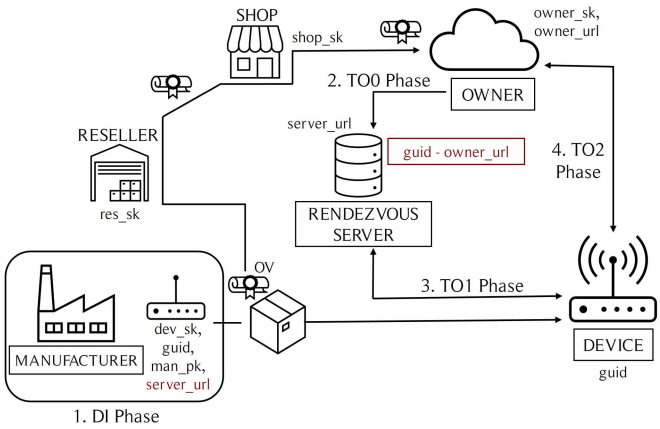


Fig. 1. FDO general scheme.

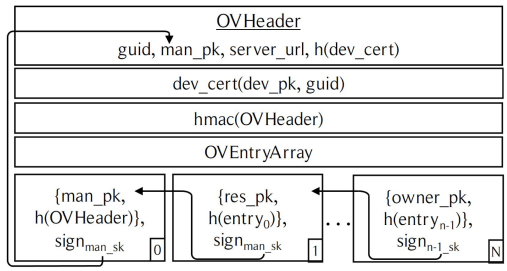


Fig. 2. Ownership Voucher structure.

shared with them. For this analysis, we employed a symbolic model and the automatic verifier Proverif [15]. All the main security properties of FDO were formally proved true, with the exception of a single scenario, in which we found a possible vulnerability under the assumption of an untrusted device supply chain.

The possibility of similar issues had already been notified by FIDO, in an app note [10], before [9] was published. In that paper, FIDO showed a slightly different possible attack, starting from the same assumptions as [9]. In the same document, FIDO also proposed some countermeasures to mitigate the attack. In Section VI, we will demonstrate that these countermeasures may not be sufficient to prevent such attacks.

Another work in the literature that also guessed potential issues for the FDO protocol is [16]. This paper mentioned the risk for the device owner of being 'at the mercy of a third-party infrastructure' due to the FDO's over-reliance on the supply chain. However, it did not describe any specific attack against the FDO in detail, nor did it explain potential solutions to be added to mitigate the guessed issue. Instead, the work proposed a completely different protocol, which can be considered an alternative to the FDO.

Another paper [17] proposed an improvement to the FDO specification focusing on privacy in edge, fog, and cloud environments. However, the formal analysis of privacy properties is completely a world apart and outside the scope of this paper. It could be considered for future work. No other works formally analysing the FDO protocol and proposing solutions to improve its security emerged from our study of the literature.

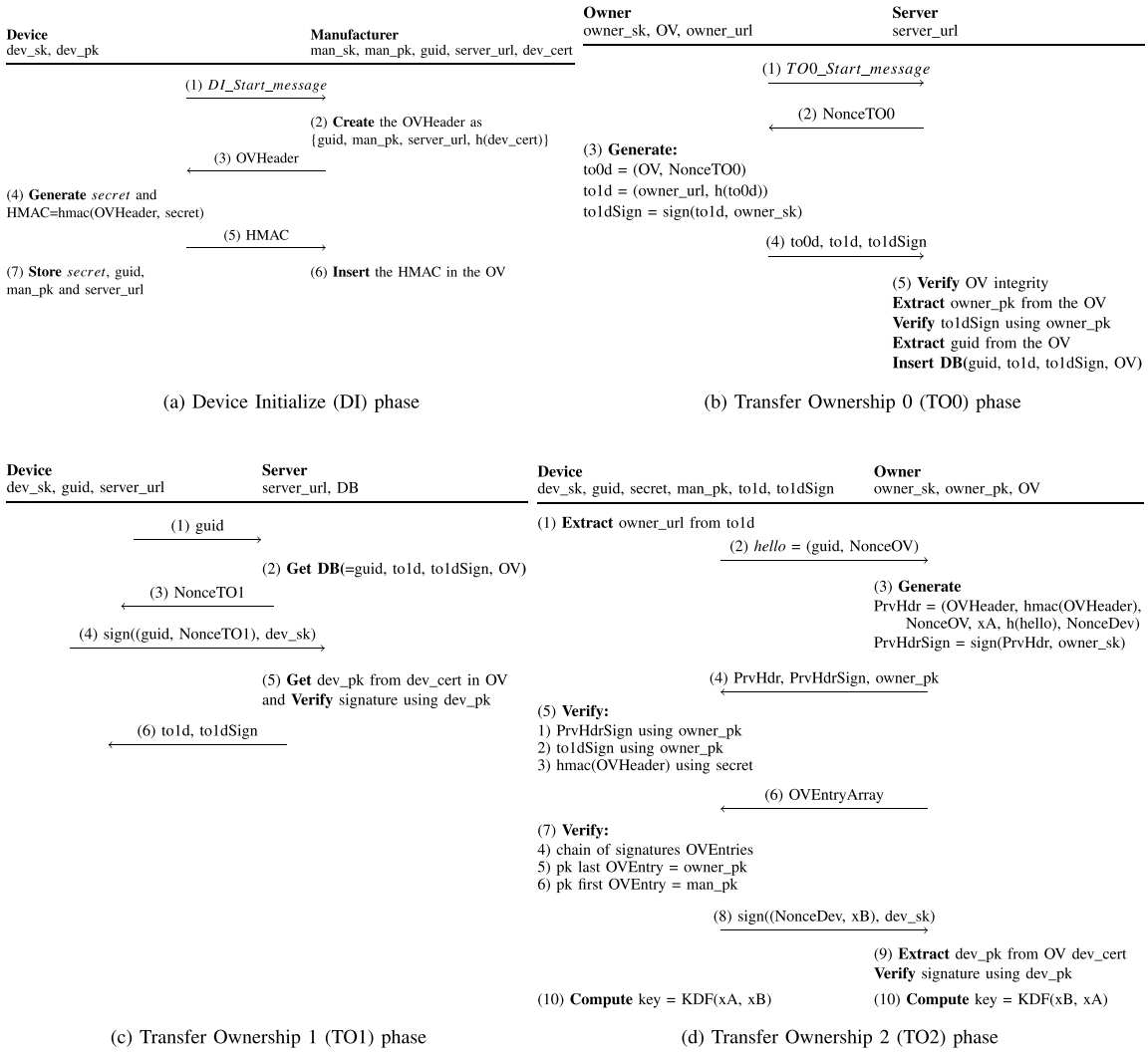
III. FIDO Device Onboard

This section provides a detailed description of how the FDO protocol operates, using version 1.1 of the specification as a reference [8].

The protocol in a nutshell: The general scheme of the protocol is shown in Fig. 1. The protocol consists of three main actors: 1) a manufacturer that builds devices, 2) a device with its own unique identifier, $guid$, and 3) a platform (also called Owner Platform or simply Owner) on which the device should be onboarded once it is turned on. The Owner waits for a connection from the device to a specific URL $owner_url$. As stated, FDO enables the device to autonomously discover this URL, without requiring manual entry or preconfiguration of this information inside the device during manufacturing. For this purpose, FDO introduces a fourth entity, the Rendezvous Server, which serves as a meeting point between the Device and the Owner. In the DI phase of the protocol, during manufacturing, the device is initialised with its secret key dev_sk, its $guid$, the manufacturer's secret key man_sk, and the URL of a rendezvous server $server_url$, to be contacted at power up. Before turning the device on, the Owner contacts the Rendezvous Server and asks the server to store the $owner_url$, on which it is waiting for a connection, and to associate it with the $guid$ of the device (TO0 phase). Once turned on, the device contacts the server to retrieve the URL associated with its $guid$, and the server returns $owner_url$ (TO1 phase). At this point, the device, holding the Owner's URL, can connect directly to the Owner and initiate automatic onboarding (TO2 phase). During the onboarding phase, it is essential to achieve mutual authentication between the Device and the Owner, and to establish and exchange a symmetric key to be used for encrypting all subsequent communications. To authenticate to the Owner, the Device uses an asymmetric secret key, dev_sk, which has been configured inside the device during manufacturing. Similarly, to authenticate to the Device, the Owner uses another asymmetric secret key $owner_sk$.

The Ownership Voucher: As mentioned, during its life cycle, the device may change its owner several times before reaching the Owner Platform. Fig. 1 illustrates an example where the device is initially purchased by a reseller, then by a general store, and ultimately by the final owner. To keep track of all these transfers of ownership, a specific certificate called the Ownership Voucher (OV) is used, which allows for the identification of the current owner of a device at a given time. A simplified structure of the OV is shown in Fig. 2. The Ownership Voucher contains:

- dev_cert: this is the X.509 certificate binding the $guid$ to the public key of the device, dev_pk, whose corresponding private part, dev_sk, was injected inside the device during manufacturing and stored in a tamper-proof memory. The device uses dev_cert and dev_sk to authenticate to the Rendezvous Server and the Owner.
- $OVHeader$: this is the header of the OV, containing the device unique identifier, $guid$, the public key of the manufacturer which built the device, man_pk, the url of



(a) Device Initialize (DI) phase

(b) Transfer Ownership 0 (TO0) phase

(c) Transfer Ownership 1 (TO1) phase

(d) Transfer Ownership 2 (TO2) phase

Fig. 3. FDO protocol phases

the Rendezvous Server which serves the requests for that device, $server_url$, and a hash computed on the device X.509 certificate, $h(dev_cert)$.

- $hmac(OVHeader)$*:* this is an HMAC computed on the $OVHeader$ using a key $secret$ known only to the device. The device keeps this secret stored internally in a tamper-proof memory and recomputes the HMAC whenever it receives the OV, to check its integrity.
- $OVEntryArray$*:* this is an array of $OVEntry$, each one identifying a specific node in the device supply chain. Each time the device is sold, a new entry is appended to the array to identify the new owner.

Each $OVEntry$ contains the public key of a specific owner who, at a given time, purchased the device from a previous owner. This public key is signed by the previous owner to certify the transfer of ownership. Each $OVEntry$ also contains the hash of the previous entry, signed by the same signature to ensure data integrity. The first entry contains the hash of the OVHeader. Initially, the device belongs to the manufacturer, so $entry_0$ contains the manufacturer's public key, man_pk. This is the same key contained in the OVHeader and the entire entry is signed by the manufacturer's private key, man_sk. Following the example of Fig. 1, the device is sold to a reseller. A new entry is therefore added, $entry_1$, containing the reseller's public key, res_pk, and the hash of $entry_0$. The entire $entry_1$ is then signed by the manufacturer (the previous owner) using man_sk, which is the private key corresponding to the public key contained in the previous entry. As said, this entry is used to certify the transfer of ownership to the reseller, who becomes the new owner of the device. Each entry is added step by step as the device progresses up the supply chain until it reaches the final owner, which will get the public key $owner_pk$ inserted in the last entry. The Owner will then use the OV and $owner_sk$ to authenticate to the Rendezvous Server and to the Device as the rightful owner.

The protocol in more detail: The various phases of the protocol are described in more detail in the next subsections. Fig. 3 can be used as a reference.

A. Device Initialize (DI) Phase

The manufacturer creates the OV and initialises the device with all the information needed to execute the FDO protocol (Fig. 3(a)). Initially, the device internally contains only an

asymmetric key pair, (dev_sk, dev_pk), created when the device is manufactured. Similarly, the manufacturer initially holds its own key pair (man_sk, man_pk). Moreover, before starting the DI phase, the manufacturer chooses a $guid$ for the device, gets a certificate dev_cert for the device from a CA, which links the device public key with its guid, and identifies the Rendezvous Server that will handle the requests from the device and the URL $server_url$ that the device will use to contact the Rendezvous Server. It is reasonable to assume that there is an agreement between the manufacturer and the server, as manufacturers choose the servers for their devices. Then, during the DI phase:

- Step 1. The device starts the protocol with a message.
- Step 2, 3. The Manufacturer creates the $OVHeader$ and sends it to the Device.
- Step 4. The Device generates a $secret$ and uses it as a key to compute the HMAC on $OVHeader$.
- Step 5. The HMAC is returned to the manufacturer.
- Step 6. The manufacturer inserts the HMAC into the OV. This HMAC will be used by the device in later stages of the protocol to verify the integrity of the OV.
- Step 7. The device internally stores the $secret$ and some of the information contained in the OVHeader (i.e., $guid$, man_pk, $server_url$) for future use.

B. Transfer Ownership 0 (TO0) Phase

The Owner registers its $owner_url$ on the Rendezvous Server (Fig. 3(b)). After purchasing the device, the Owner obtains the OV with its own public key, $owner_pk$, contained in the last entry of the $OVEntryArray$.

- Step 1. The Owner extracts $server_url$ from the OVHeader and sends a Start message to the server.
- Step 2. The server replies with a nonce, $NonceTO0$, that the Owner must sign with $owner_sk$ to demonstrate proof of ownership of the OV.
- Step 3. The Owner computes: **(i)** $to0d$, as the received $NonceTO0$ and the OV, used in this phase by the Owner to demonstrate proof of possession of the OV; **(ii)** $to1d$ as $h(to0d)$ (the hash of $to0d$) and $owner_url$, which will be returned to the device in phase TO1; **(iii)** $to1dSign$, which is $to1d$ signed by the Owner using $owner_sk$.
- Step 4. The server receives $to0d$, $to1d$, and $to1dSign$.
- Step 5. The server **(i)** extracts the OV from $to0d$ and verifies its integrity (checks the chain of signatures in the $OVEntryArray$,[1] checks the hash of the OVHeader is equal to the value contained in the first entry of the OVEntryArray); **(ii)** extracts $owner_pk$ from the last entry of the OVEntryArray, and uses this key to verify the signature on $to1dSign$.[2] If all these checks are correct, the Owner has been correctly authenticated. At this point, the server **(iii)** extracts the $guid$ of the device from the OV and **(iv)** inserts an entry in its DB associating $guid$ with $to1d$, $to1dSign$, and OV.[3]

At the end of this phase, the Owner starts waiting for a connection from the device to the specified $owner_url$.

C. Transfer Ownership 1 (TO1) Phase

The device retrieves the owner_url from the Rendezvous Server (Fig. 3(c)).

- Step 1. The device sends its $guid$ to the server.
- Step 2, 3. The server extracts the entry associated with $guid$ from its DB and sends a nonce, $NonceTO1$, which the device must sign to authenticate itself.
- Step 4. The device signs $NonceTO1$, together with its $guid$, using dev_sk and sends the signature to the server.
- Step 5, 6. The server retrieves dev_cert from the OV extracted from its DB, and uses the dev_pk in dev_cert to verify the signature received from the device. If the signature is valid, then the device has been correctly authenticated and can receive $to1d$ and $to1dSign$.[4]

D. Transfer Ownership 2 (TO2) Phase

Device and Owner perform mutual authentication and exchange the symmetric session key (Fig. 3(d)). This is the main phase of the protocol.

- Step 1, 2. The device extracts $owner_url$ from the $to1d$ received in the TO0 phase and sends a $hello$ message to the Owner, containing the device $guid$ and a nonce, $NonceOv$, that the Owner must sign to prove ownership of the OV.
- Step 3. The Owner generates **(i)** $PrvHdr$, containing the $OVHeader$, the HMAC included in the OV, the received $NonceOv$, the first parameter for the key exchange algorithm, xA, and a nonce, $NonceDev$, which must be signed by the device for mutual authentication; **(ii)** $PrvHdrSign$, which is $PrvHdr$ signed using $owner_sk$.
- Step 4. The Owner sends $PrvHdr$, $PrvHdrSign$, and $owner_pk$ to the device.
- Step 5. The device performs the following checks: **1)** it verifies $PrvHdrSign$ using the received $owner_pk$; **2)** it verifies the stored $to1dSign$ using again $owner_pk$; **3)** it recomputes the HMAC on the OVHeader using $secret$ (from the DI phase) and verifies that this HMAC matches the one received from the Owner.
- Step 6. The Owner sends OVEntryArray to prove to the device that it is the legitimate owner. $owner_pk$, which the device used to perform the previous checks at Step 5, should be the last key in OVEntryArray.
- Step 7. The device verifies **4)** the chain of signatures in the OVEntryArray, **5)** the last entry contains $owner_pk$, **6)** the

[1]To perform this check, the server uses man_pk as root of trust, relying on the agreement existing between the server and the manufacturer. Thus, it is assumed that the server already knows man_pk (and checks that this is the actual key contained in the OVHeader), and uses it to verify the signature on the first entry of the OVEntryArray.

[2]This check verifies that the Owner actually possesses $owner_sk$ and, consequently, is the legitimate owner of the OV. This signature check indirectly also covers $to0d$, because of the reference $h(to0d)$ contained in $to1d$, and, therefore, the nonce $NonceTO0$ initially sent by the server.

[3]$to1d$ is particularly important because it contains the $owner_url$ that will be used by the device to find the owner to connect to.

[4]The device cannot verify $to1dSign$ yet because it does not know the $owner_pk$. It will perform the verification in the TO2 phase after receiving the OV from the Owner.

first entry contains man_pk.[5] If all checks are passed, the Owner has been correctly authenticated.

- Step 8. To authenticate itself and complete mutual authentication, the device signs $NonceDev$ using dev_sk. The signature also includes xB, the second parameter for the key exchange protocol.
- Step 9. The Owner extracts dev_pk from the dev_cert contained in the OV, and verifies the signature on the received message. If the signature is valid, the device has correctly authenticated itself.
- Step 10. Both nodes have xA and xB and can complete the key exchange protocol and generate the symmetric session key, to protect all subsequent communications.

Concerning the key exchange protocol, FDO supports a wide range of algorithms. In the example above, we could assume the use of an authenticated Diffie–Hellman key exchange, where xA and xB denote the two exponents.

As a final note, regarding secure communications between the nodes, FIDO in [13] additionally recommends the use of HTTP over TLS in all the protocol phases, as a further protection mechanism (as indicated in Table I). However, from the specification, this is more like a suggestion than a mandatory requirement. In [8], in fact, the authors write that the device can use "either HTTP or HTTPS", thus suggesting the use of https only for devices that have the capability and can support it. In this paper, we verified the version of the protocol providing strong authentication, confidentiality, and data integrity at a higher layer, for the cases where HTTPS is not used.

IV. Threat Model

This section presents the threat model that was used to guide our analysis. It considers the most significant threats that were identified by the FIDO Alliance in the comprehensive threat model they developed [13]. The threat model is described by specifying and motivating the main threat assumptions that have been made and the main threat scenarios.

A. Assumptions About Protocol Actors

In our analysis, we assume the existence of a number of legitimate, designated, and honest protocol actors, covering all the protocol roles: (i) FDO-capable devices, (ii) Manufacturers, (iii) Manufacturer-selected Rendezvous Servers, and (iv) Owners. We assume that such legitimate actors cannot be compromised; i.e., the attacker cannot obtain their private keys and thus cannot directly act on their behalf (Assumption 1). Were these private keys compromised, the FDO security properties would be violated by design, since the FDO specification does not guarantee any security property in the event that legitimate entities are compromised. Conversely, we assume the attacker can control one or more dishonest protocol entities, using only their own (attacker's) keys. More precisely, the attacker can impersonate Devices, Rendezvous Servers, or Owners during the various phases of the protocol. On the other hand, we assume the attacker cannot impersonate Manufacturers (Assumption 2). This assumption descends from the assumption made by the FIDO Alliance that Manufacturers are always honest and that devices are produced in a secure and uncompromised environment (Assumption 3). Protecting device integrity during manufacturing, as well as any subsequent verification procedure, is considered out of scope for the FDO protocol.

[5]The device stored man_pk during the DI phase. It is now used as root of trust to validate the chain of signatures in the OVEntryArray.

B. Assumptions About Communication Channels and Attacker Capabilities in the Various Protocol Phases

Almost all phases of the protocol occur over a public network. This implies that, when the protocol is not protected by HTTPS, the attacker has full control over the exchanged messages, and their security relies solely on the cryptographic mechanisms defined by FDO. Therefore, the attacker can perform any attack feasible on an untrusted network such as sniffing messages or deleting, modifying or forging messages, or performing cryptographic attacks on them. The only exception, as mentioned earlier, is the DI phase, which takes place in a secure environment (i.e., within the Manufacturer's facility), where the attacker cannot act (Assumption 4). In contrast, the attacker can interfere with and potentially compromise the TO0, TO1, and TO2 phases.

C. Assumptions About the Ownership Voucher

The FDO specification explicitly states that the OV does not reside in the device secure hardware, but it is handled externally by design. Its storage is delegated to the entities participating in the supply chain, and the standard does not impose strict requirements on how this storage must be implemented. Therefore, the OV can be considered publicly accessible information, and its security relies exclusively on the cryptographic mechanisms described above. In other words, an attacker may have full control over the OV's storage and handling. However, to successfully perform attacks, an attacker must evade the cryptographic checks enforced by the protocol to protect the OV.

D. Threat Scenarios

After having presented the general assumptions underlying the threat model, we identify three different threat scenarios to be studied, characterised by increasing attacker's capabilities:

1) *External attacker:* The attacker is not part of the protocol. It attempts to compromise FDO from the outside by manipulating the messages exchanged among the legitimate entities. In this scenario, both the supply chain and the device owners are assumed to be fully trusted.
2) *Dishonest device owner:* In addition to the capabilities of the previous scenario, the attacker owns valid devices that can be onboarded on the platform. In this second scenario, the device's supply chain is still trusted, but some device owners are malicious and may try to compromise the onboarding of other devices.
3) *Untrusted supply chain:* In addition to the capabilities of the previous scenario, the attacker may also be an intermediate node in the supply chain, which legitimately

TABLE II
MAIN THREATS CONSIDERED IN OUR VERIFICATION

Protocol phase	ID	Threat description
DI phase	-	... Assumed secure ...
TO0 phase	TH01	MitM - Server impersonation to the Owner
	TH02	MitM - Owner impersonation to the Server
	TH03	Replay Attack and traffic manipulation (TO0)
	TH04	DOS on the Server
TO1 phase	TH05	MitM - Server impersonation to the Device
	TH06	MitM - Device impersonation to the Server
	TH07	Replay Attack and traffic manipulation (TO1)
	TH08	DOS on the Server
TO2 phase	TH09	MitM - Device impersonation to the Owner
	TH10	MitM - Owner impersonation to the Device
	TH11	Replay Attack and traffic manipulation (TO2)
	TH12	Disclosure credentials (symmetric session key)
	TH13	Eavesdropping of messages exchanged after TO2
	TH14	DoS (unusable device)
	TH15	Full control of the device

buys the device and subsequently resells it to another node. By doing so, the attacker can be the rightful owner of the device for a limited time, having its own public key within the entries of the OV.

Based on the information reported in [13] and the assumptions discussed above, the main threats to be considered are those listed in Table II.

V. ATTACKS AGAINST THE OV

The specification of the protocol as described in Section III is subject to a security issue when the attacker is an intermediate node in the supply chain (third scenario of our Threat model). This issue was identified and described in our previous paper [9] and in the app note [10].

Let us imagine, for instance, that the attacker buys the device from node $n-1$ in the supply chain and, later, resells it to node $n+1$. The attacker's att_pk is correctly inserted in $entry_n$ of the OV, legitimately signed by node $n-1$. By selling the device to node $n+1$, the attacker adds $entry_{n+1}$, containing the new $owner_pk$, signed using att_sk.

The attacks described in [10] and in [9] are due to the fact that, once the attacker has its key inserted in $entry_n$, it has full control over the entries that follow, i.e., $entry_i$ where $i > n$. The attacker can simultaneously create different OVs with different pk values in their respective $entry_{n+1}$. For example, the attacker can create **1)** OV(a) with $owner_pk$ in $entry_{n+1}$, transferring ownership to a legitimate new owner who has purchased the device from the attacker, and **2)** OV(b) with $att2_pk$ in $entry_{n+1}$, transferring ownership to a second malicious owner. The attacker can even use the pre-extended version of the OV, as received from node $n-1$, and leave it with n entries. We call this, **3)** the truncated version of the OV, or OV(c), having att_pk in the last entry, $entry_n$.

All the three OVs (i.e., OV(a), OV(b), and OV(c)) can be used successfully by their specified owners to authenticate. In fact, in all cases, the chain of signatures is verified correctly. In a realistic case, the device is purchased by a legitimate owner, who has physical control over it and owns OV(a). However, the attacker still manages to authenticate itself as the legitimate owner - even after selling the device (as if the sale had never occurred) - by using either OV(b) or OV(c).

This vulnerability is possible because, when authenticating the Owner, both the Device and the Rendezvous Server have no way of knowing how many (and what) transfers of ownership have occurred in the supply chain of the device. To authenticate the Owner, both nodes simply extract the public key contained in the last entry of the OV (which they consider to be the key of the legitimate owner) and verify the chain of signatures. This last key could belong to the attacker, as $att2_pk$ or att_pk in cases OV(b) and OV(c), when the attacker is an intermediate node in the supply chain. The OV provides a mechanism to protect the insertion of unauthorised new entries, but because of this vulnerability, it does not include ways for verifying whether some entries have been removed or if the OV has been extended multiple times for multiple recipients at the same time.

Example scenario
To simplify the reader's understanding, Fig. 4 shows a practical example of this vulnerability, where the attacker purchases the device from the manufacturer and then resells it to a new legitimate owner. The versions of OV(a), OV(b) and OV(c) in this case are shown in Figs. 4(a), 4(b), and 4(c).

A. Consequences

As a consequence of the issue described above, the attack can proceed in the different phases of the protocol as shown in Fig. 4(d).

1) Attack in the TO0 phase: The attacker, using a malicious OV, e.g. OV(b) (but it could also be OV(c)), succeeds in authenticating to the Rendezvous Server as the legitimate owner of the device (TH02). It can use its $att2_sk$, whose corresponding $att2_pk$ is contained in the last entry of OV(b), to sign the $NonceTO0$ sent by the server. In this case, the attacker manages to convince the server to store its att_url associated with the $guid$ of the device. This URL will then be returned to the device in the TO1 phase, so the device will never attempt to open a connection with the real Owner, leading to a denial of service attack (TH14).

Note: Since even the real Owner has a valid OV, OV(a), that can be used to authenticate to the server, a few more considerations should be added. The FDO specification [8] states that the TO0 phase can be executed multiple times, with each new request to the server overwriting the old value stored in its database with a new association. This is enabled to give the Owner the ability to change the connection URL. For this reason, to succeed, the attacker should periodically repeat the TO0 phase to maintain the attack state, in the event that a legitimate request has arrived from the real Owner (overwriting the entry containing att_url with $owner_url$). If the server does not perform additional checks on the OV used by the requesting node to authenticate, but merely extracts and uses the public key from the last entry without checking if it is consistent with any previous requests, this is a possible attack. Alternatively, the attacker could block the connection between the Owner and the Server by building a Denial of Service attack against the Owner, to prevent it from sending the request.

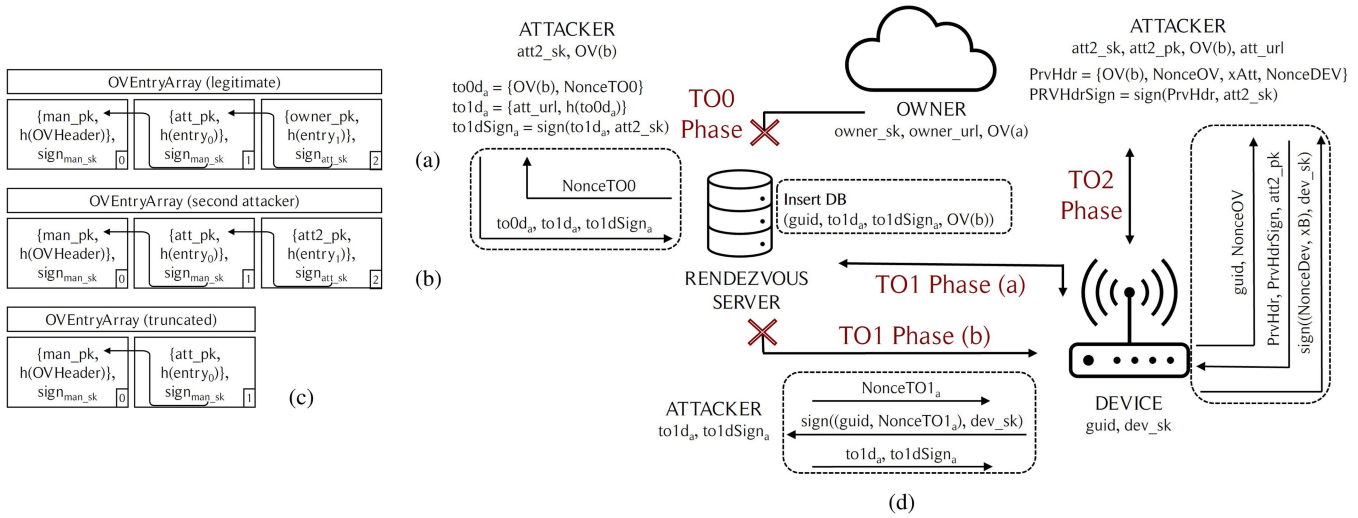


Fig. 4. Attacks against the OV: general scheme.

2) Attack in the TO1 phase: Two scenarios may happen in the TO1 phase.

1) If the attacker succeeded in storing its $to1d_a$ containing att_url on the server during the TO0 phase, now, it simply has to wait for the device to retrieve att_url from the server and connect to it (Fig. 4(d), TO1 Phase a).
2) Otherwise, the attacker can mount a MitM attack between the Device and the Rendezvous Server. Since the TO1 phase does not involve server authentication, if HTTPS is not used to secure communication, the attacker can insert itself between the device and the server and return $to1d_a$ and $to1dSign_a$ to the device performing the TO1 phase (TH05). $to1d_a$ contains att_url and the signature on $to1dSign_a$ is performed using $att2_sk$ (Fig. 4(d), TO1 Phase b).

In both scenarios, the device will contact the attacker at att_url in phase TO2. The attacker will use OV(b) and $att2_sk$ to demonstrate to be the rightful owner of the device (TH10).

3) Attack in the TO2 phase: The device, which has received $to1d_a$ and $to1dSign_a$ in the TO1 phase, contacts the attacker at att_url and performs mutual authentication and symmetric key exchange with the attacker (TH12). To authenticate to the device, the attacker signs $PrvHdr$ using $att2_sk$, which is the same key used to sign $to1dSign_a$ and whose corresponding public part $att2_pk$ is contained in the last entry of the OV(b) sent to the device. After verifying the correctness of the chain of signatures in the OV (thus completing the attacker authentication), the device executes the key exchange protocol by sending the parameter xB to the attacker. Both the attacker and the device can now derive the symmetric key (KDF($xAtt$, xB)). With this key, the attacker has full control of the device (TH15).

Note: The real Owner who purchased the device from the attacker and got it physically does not detect the attack. After turning the device on, no connection is established to $owner_url$, because the device has connected to att_url. From the Owner's perspective, the device simply appears not to be working.

B. Reproducing the Attacks on the Reference FDO Implementation

The attacks described above, along with the related consequences, have been tested on a real implementation of the protocol, i.e. the one released by the FIDO Alliance via the Linux Edge Foundation, available at the open-source repository [18]. The repository contains a production-ready implementation of all the actors involved in the protocol, along with a demo to show the step-by-step execution of the protocol. Individual components are provided either as standalone Java applications or as Docker containers, and they share a unique MariaDB database, in the form of a Docker container, for persistent storage. The implemented components, apart from the Device, are the Rendezvous Server, the Manufacturer, the Owner, and the Reseller. For each of them, various HTTP/HTTPS APIs are provided to invoke the different services offered by the entities. For example, there is an API to retrieve the owner certificate, one to request the extension of the OV, and another to initiate the execution of a specific phase of the protocol. By using these APIs, it is possible to run a customised demo to inspect different aspects of the protocol.

We performed several tests on the implementation. Below, we present the general idea behind each test. A pdf file with a more detailed description of all the steps performed during the tests is available at https://github.com/netgroup-polito/verification-fdo.

Test 1: In the first test, we tried to create two distinct OVs (OV(a) and OV(c) from Fig. 4) to be used to authenticate the Owner in the different phases of the protocol. The goal of this test is to try to replicate the attack described in our previous paper [9], in which a malicious node in the supply chain manages to authenticate itself as the legitimate owner of the device, even after selling it. To this purpose, we used the APIs provided by FIDO and we slightly changed the demo described in [18] by adding additional steps. Initially, we used the APIs to issue a valid OV to the Reseller, which, in our tests, acts as the attacker. This step represents the transfer of the device ownership from the Manufacturer to the Reseller, which becomes a valid node

in the supply chain. Once obtained the valid OV for the Reseller (returned in the body of the HTTP response, and corresponding to OV (c)), we stored a copy of this certificate, and we used the APIs to extend it for the Owner, obtaining in this way also OV(a). The API to extend the OV requires, as a parameter, the certificate containing the public key of the future new owner of the device, and in this case, we specified the Owner's certificate (available by calling an Owner API). Having reached this point, we tried to execute the TO0 phase of the protocol with both OV(a) and OV(c). There is a specific Owner API to invoke to start the TO0 phase with the Rendezvous Server. Following the demo, once this API is called, the process takes place automatically, and it uses, as parameters, the information stored within the MariaDB database. The call to this API, in fact, retrieves the OV stored in the DB table *onboarding_voucher* and the keys stored in *certificate_data* under the row associated with the Owner, as the data to use to authenticate the requester node to the Rendezvous Server. Therefore, by acting directly on the database and changing the values stored on these tables, it is possible to influence which data is used by the node to authenticate to the server. In our demo, we called this API two times: the first time with OV(a) and the correct keys of the Owner stored in the DB; the second time with OV(c) and the keys of the Reseller (i.e., our attacker). In both attempts, the call was successful, and it correctly authenticated the node as the rightful owner of the device. With this test, we succeeded in reproducing the attack found in [9] using only the APIs provided by FIDO for the demo, without the need to write or modify any component application code.

Test 2: In the second test, we tried to perform a MitM attack in the TO1 phase between the Rendezvous Server and the Device. The APIs provided for the demo alone are not sufficient to perform this attack, and new application code must be written. To this purpose, we built a malicious Java server accepting requests from the device at the same IP addresses and ports of the legitimate Rendezvous Server. The malicious server is programmed to return to the requesting device a $to1d_a$ entry containing the attacker's $attacker_url$ and a $to1dSign_a$ signed with the private key of the Resellers (which, in our tests, acts as the attacker). Since there is no server authentication in the TO1 phase, the returned values are accepted by the device as coming from the legitimate server, so the device stores $attacker_url$ as the URL to contact in the subsequent TO2 phase. To show this, we ran the same demo as the previous test, but with our malicious server replacing the real Rendezvous Server in the TO1 phase. In particular, the DI and TO0 phases remained the same as in the original demo described by FIDO, with the Manufacturer and Device performing the DI phase, and the legitimate Owner completing the TO0 phase with the real Rendezvous Server. Then, for the TO1 phase, we switched off the real server and turned on our malicious one, simulating a traditional MitM attack. We found that the device successfully accepted $to1d_a$ and $to1dSign_a$ returned by the malicious server as if they came from the real Rendezvous Server.

Test 3: In the third test, we tried to perform a MitM attack in the TO2 phase between the Owner and the Device. To this purpose, we built a second Java Server acting as a malicious Owner and waiting for a connection from the device at the IP address and port specified in $attacker_url$. We assigned this malicious owner the public and private keys of the Reseller (i.e., our attacker) and OV(c) to authenticate to the device. The device, which received $to1d_a$ and $to1dSign_a$ from the hacked TO1 phase, contacts our malicious owner at $attacker_url$. The malicious owner authenticates to the device using OV(c) and $reseller_sk$. Recall that OV(c) contains the $reseller_pk$ in the last entry, and that this is the key that correctly verifies the signature done here in the TO2 phase by our malicious owner and the one on $to1dSign_a$ from the previous phase. What we found is that the device and the malicious owner using these credentials automatically completed mutual authentication and correctly performed a symmetric key exchange, as expected from a correct execution of the protocol.

With these three tests, we proved that the attacks described at the beginning of the section are also possible against the actual implementation of the protocol. Starting in the next section, we analyse some solutions that can be implemented when trying to solve the issue.

VI. Possible Countermeasures

Several countermeasures have been proposed by FIDO to mitigate the attacks described in Section V. Below, we comment on some of the solutions described in the app note [10]. Although these solutions can make the attacker's work much more difficult, in some cases, they do not completely solve the security issue.

1) The most trivial solution to adopt is to verify, after the device has been turned on, that it actually onboards on the correct platform as planned. For instance, if the device does not contact the real Owner within a certain time, an attack may be in progress, so the Owner can physically remove the device and retry the entire FDO protocol from the beginning. This solution, although very simple, has obvious scalability limitations. In fact, if the number of devices to be onboarded is large, checking one by one which devices have correctly completed the FDO protocol and finding those that have not been successfully onboarded can be time-consuming. Furthermore, this is valid only if we assume that a single user controls both the device and the Owner Platform. If the platform is a third-party service not joined with the company that owns the device, challenges such as timely communication and synchronisation between the two parties also have to be considered.

2) A second, more convincing, countermeasure is to add checks on the Rendezvous Server, especially in the TO0 phase, when the server receives the OV from the node which claims to be the owner of the device. Since the TO0 phase can be executed multiple times, with each new request overwriting the value inserted by the previous one, the server can perform more accurate checks on the OV received to authenticate the requesting owner. The OV must be the same in all the requests. For example, if the server receives a first request from the real Owner, which uses the OV(a) of Fig. 4(a) to authenticate, subsequent requests from the attacker using either the OV(b) of Fig. 4(b) or the OV(c) of Fig. 4(c) shall be blocked (because they

are using an OV different from the OV(a) of the first request). In particular, the app note [10] proposes the following additional actions on the server:

- Once the server has completed a first run of the TO0 phase with a prospective owner, it shall cache the owner's public key contained in the last entry of the OV used for authentication.
- The server shall then reject all subsequent TO0 requests from owners that send an OV with the last key that does not match the cached one
- An attempt to run the TO0 phase with a different key shall return an error message

With this countermeasure, only one node can perform, for an indefinite number of runs, the TO0 phase with the server. The Owner, in turn, before turning on the device, can verify that the key cached on the server matches its own public key ($owner_pk$), to be sure that the server has not run the TO0 phase with the attacker. In this way, it is sure that the server will return $owner_url$ to the device, as soon as the device is turned on in the TO1 phase. Conversely, the attacker is not able to hack the protocol because, despite having a valid OV associated with the guid of the device (either OV(b) or OV(c)), the key cached on the server does not match the key in the last entry of that OV (respectively $att2_pk$ or att_pk). The server would then reject its requests to run the TO0 phase. On the other hand, any attempt from the attacker to use OV(a) of Fig. 4(a) to authenticate to the server (i.e., the only OV accepted containing $owner_pk$ in the last entry) would fail because the attacker does not know $owner_sk$, so it cannot execute the signature on $to1dSign$.

Results: Although promising, these additional checks do not resolve the denial of service attack in the case that the attacker is the first to execute the TO0 phase with the server (before the real owner). In this case, the server caches the attacker's public key associated with the guid of the device, so it will accept TO0 phases only with the attacker, excluding basically the real owner from the protocol.

The app note [10] does not specify how the Owner can resolve this situation, nor with what protocol it can query the server to check which public key it has cached.

A. MITM Attack Possible With the Proposed Countermeasure

We performed a formal analysis of the protocol with the countermeasure presented in the previous section, using Proverif. This analysis revealed a MitM attack is still possible.

The attack scenario found with the analysis occurs after the real Owner has executed the TO0 phase with the Rendezvous Server. As discussed, in this case the attacker is not able to associate its att_url to the guid of the device because it does not have a valid OV for the key cached on the server. Therefore, the TO0 phase cannot be hacked and the server will always return correct entries from the real Owner.

Issue: In this situation, there is still a possible attack, which is a MitM between the device and the server in the TO1 phase, i.e. the second scenario described for the *"Attack in the TO1 phase"* in Section V-A, shown in Fig. 4(d) (TO1 Phase b).

This attack is described in Fig. 5 for better understanding. Since there is no server authentication in the TO1 phase, if HTTPS is not used to authenticate the parties (and this is possible since HTTPS is not mandatory for devices in the specification), the attacker can still interpose itself between the device and the server, and return its entry, $to1d_a$, with att_url, to the device. Therefore, even if the Rendezvous Server has correct entries because it did not perform the TO0 phase with the attacker, it would never be contacted by the device, which, in turn, receives the malicious entry from the attacker (TH05). The device will then perform the TO2 phase with the attacker by contacting it at att_url (TH10).

Considerations: Based on these results, we strongly believe that any countermeasure that strengthens controls on the Rendezvous Server, to be effective, must necessarily consider server authentication in all the phases of the protocol in which the server is involved.

VII. Our Proposed Solution

In this section, we propose a possible solution that can definitely prevent all the attacks we found affecting the FDO specification. In particular, we propose some minor changes to the specification, introducing server authentication, even when HTTPS is not used (particularly important for device-server communications, where HTTPS is not always feasible).

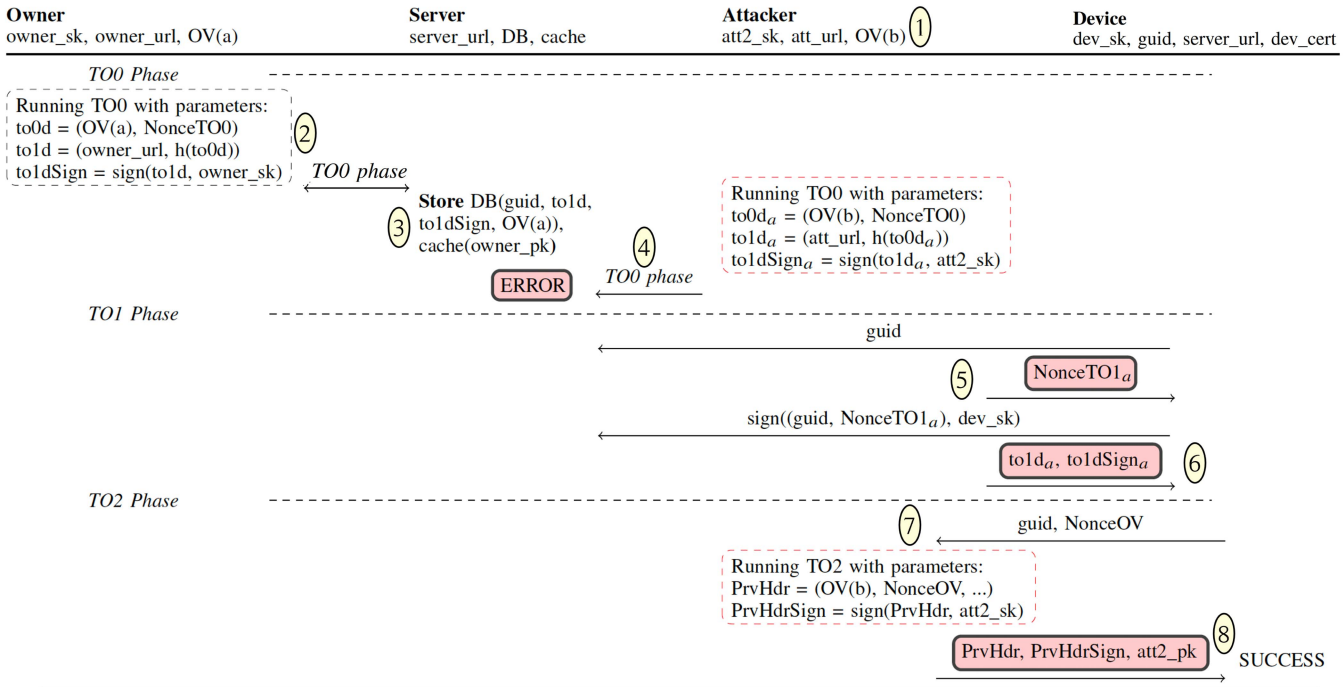
For this purpose, we propose to equip the Rendezvous Server with a certificate that can be used in phases TO0 and TO1, to authenticate respectively to the Owner and the Device. Specifically, the Rendezvous Server is assigned a pair of public-private keys, $server_pk$ and $server_sk$, and a certificate, $server_cert$, obtained from a CA, containing $server_pk$. A hash of that certificate is then inserted into the OVHeader, which has the new structure shown in Fig. 6.

The new OVHeader contains both $server_url$ and $h(server_cert)$, i.e., the hash of the certificate that the server will use, together with a signature, to authenticate itself.

The protocol phases change as shown in Fig. 7 and as described in the following sections. A proof of correctness, showing that this solution prevents all the attacks described in Section V, has been obtained by means of Proverif, as detailed in the next section.

A. Modified DI Phase (Fig. 7(a))

The only difference in this phase, compared to the original protocol described in the specification, is that now the manufacturer, after choosing the server that will handle requests for a device, must get both $server_url$ and $server_cert$ from that server. Then, the manufacturer computes $h(server_cert)$, and inserts it into the OVHeader. The device performs the same operations as in the original protocol: it generates a $secret$, computes the HMAC on the received OVHeader (which now also contains $h(server_cert)$) using the secret, and stores $secret$, $guid$, man_pk and $server_url$ in a secure storage.



Attack trace	
1. The attacker is part of the supply chain of the device and has a valid OV(b), like the one in Fig. 4b.	2. The real Owner is the first to execute the TO0 phase with the Server. It uses its OV(a) from Fig. 4a to authenticate.
3. The Server stores the Owner's entry and caches the $owner_pk$. Subsequent requests to rerun the TO0 phase can only use that key.	4. The attacker tries to execute the TO0 phase using OV(b), but it gets an error. It cannot store its entry on the Server.
5. The device sends its $guid$ to initiate the TO1 phase with the Server, but the request is intercepted by the attacker.	6. The attacker replies with $to1d_a$ (att_url) to the device. Since there is no server authentication, the entry is accepted by the device.
7. The device contacts the attacker at att_url in the TO2 phase. The attacker uses OV(b) and $att2_sk$ to authenticate to the device.	8. The device checks OV(b) and verifies the signatures using $att2_pk$ (key contained in the last entry of OV(b)). All checks are ok.

Fig. 5. Attack against the countermeasure.

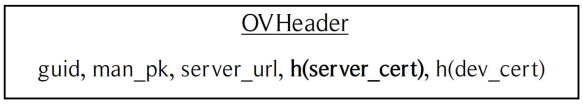


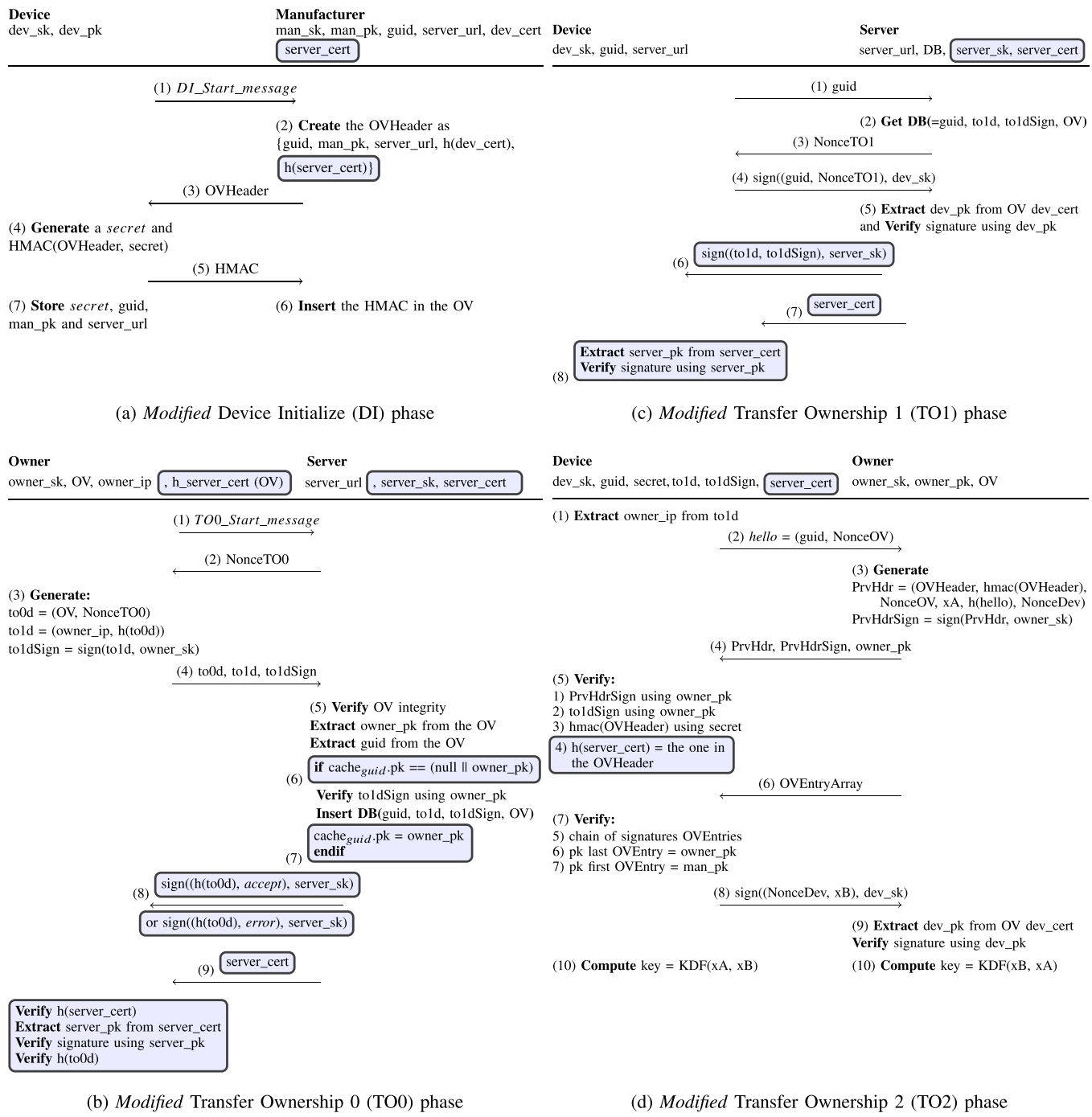
Fig. 6. Modified OVHeader.

B. Modified TO0 Phase (Fig. 7(b))

This phase is modified to obtain server authentication. The Owner, who obtained the OV on purchasing the device, can extract $h(server_cert)$ from the OVHeader. To authenticate to the server, the Owner follows the same steps as in the original protocol: after receiving $NonceTO0$ from the server, the Owner computes $to0d$ and $to1d$, and then the signature on $to1dSign$ using $owner_sk$ (Steps 1-4). After receiving these messages from the Owner, the server verifies the integrity of the OV contained in $to0d$. It then extracts $owner_pk$ from the last entry of the OVEntryArray and the $guid$ of the device from the OVHeader (Step 5).

Starting from this point, the new protocol differs from the original one to include the countermeasure suggested in the app note [10]. The server checks its cache to verify if it already contains a public key associated with the device $guid$. An **empty cache**, i.e., $cache_{guid}.pk == null$, means that the server never received TO0 requests for that device in the past. In this case, the server proceeds with the authentication of the requesting owner by verifying the signature on $to1dSign$ using $owner_pk$. If the verification succeeds, it inserts the received entry in the DB and $owner_pk$ in the cache. Only the owner holding $owner_pk$ will be able to re-run the TO0 phase with the server to update the entry stored in the DB. If instead the **cache is not empty**, the server verifies that the cached key matches the $owner_pk$ contained in the last entry of the received OV, i.e., $cache_{guid}.pk == owner_pk$. If the two keys match, this means that the requesting owner is the same one who already performed the TO0 phase initially. Therefore, the update of the DB entry is allowed. Otherwise, the newly received request is from an owner different from the initial one, and the request is refused (Steps 5-7).

In both cases, the server returns a response message, either an $accept$ or an $error$. This message is used by the Owner to verify whether the pk in the server cache corresponds to what expected, before turning the device on. In the case of an accept message, the Owner has assurance that the server never ran the TO0 phase with other owners, especially the attacker. The hash $h(to0d)$



(a) *Modified* Device Initialize (DI) phase

(c) *Modified* Transfer Ownership 1 (TO1) phase

(b) *Modified* Transfer Ownership 0 (TO0) phase

(d) *Modified* Transfer Ownership 2 (TO2) phase

Fig. 7. *Modified* FDO protocol phases

contained in the signed response message is also important. In fact, it uniquely associates the server response with the Owner's request, preventing replay and MitM attacks (Step 8).

Together with the response message, the server also sends its $server_cert$ (Step 9). When receiving the message, the Owner extracts $server_cert$ and computes its hash, $h(server_cert)$, to verify that it is the same value contained in the OVHeader. Then, it extracts the $server_pk$ from the certificate and verifies the signature on the message received from the server. Finally, the Owner verifies that the $h(to0d)$ received in the message is actually equal to the hash of the $to0d$ field that the Owner sent to the server at the beginning of the phase (Step 10). If all these checks are correct, the server has been correctly authenticated, and the Owner can turn the device on.

C. Modified TO1 Phase (Fig. 7(c))

The TO1 phase has also been modified compared to the original protocol to achieve server authentication. In the first few steps, the device signs the $NonceTO1$ sent by the server for

device authentication, as in the original protocol (Steps 1-5). The response from the server, on the other hand, is different, because it is now signed by the server using $server_sk$ (Step 6). In the response message, the server also attaches its $server_cert$, to allow the device to verify the signature (Step 7). After receiving the message, the device extracts $server_pk$ from $server_cert$ and uses it to verify the signature (Step 8).

Note: At this stage, the device cannot complete server authentication. Indeed, the device is not able to verify that the hash of $server_cert$ is the same value as the one contained in the OVHeader, because it does not have this information stored in memory. The device will complete server authentication with this additional check during phase TO2, after receiving the OVHeader from the prospective Owner.

Note 2: Another possible solution, which enables server authentication in this phase, is to store $h(server_cert)$ within the device during the DI phase. In this way, when the device receives $server_cert$ from the server, it can recompute this hash and compare it with the one stored in memory. However, this solution requires more memory on the device and does not add any security benefit.

D. Modified TO2 Phase (Fig. 7(d))

The TO2 phase, which does not involve the server because it takes place between the Device and the Owner, is almost identical to the one in the original protocol. Here, the only difference is that the Device, after receiving the OVHeader in the $PrvHdr$ field, extracts $h(server_cert)$ from the header and verifies that it is equal to the hash of the $server_cert$ received in the TO1 phase (Step 5). This check is used to complete the server authentication and to be sure that no MitM happened in the TO1 phase.

Note: This check must be performed by the device after verifying OVHeader integrity, by computing the HMAC on the header using the $secret$ stored during the DI phase.

The device then continues the TO2 phase by authenticating the Owner, as in the original protocol.

VIII. Formal Analysis

In this section, we describe the formal models we used to validate all the observations made in this paper.

As said, the FDO protocol has been formally verified using the automatic tool Proverif [15], which takes as input an abstract symbolic model of the protocol and the security properties that must be verified, and looks for a mathematical proof that the properties hold in the given abstraction, under certain reasonable modelling assumptions. If the proof cannot be found, an attack may be possible on the model and Proverif may return such an attack. Formal verification offers several benefits over other traditional verification methods such as dynamic analysis and testing. Most importantly, it is more rigorous because it can prove that a system model is correct with mathematical certainty. At the same time, formal verification tools like Proverif have some limitations that should be well understood and that are highlighted below.

Proverif limitations: Formal verification is typically performed on an abstraction of the protocol, built according to some modelling assumptions. In Proverif, the model is symbolic, i.e. cryptography is modelled through function symbols in an algebra of terms. This means that cryptographic operations are assumed to be ideal and attacks against their incorrect selection or application (which may happen in a real implementation) are not taken into account by the verification. This makes our Proverif model a slightly looser abstraction compared to the threat model presented in Section IV, where the attacker is assumed to be able to exploit any cryptographic weakness affecting messages exchanged over the public network as well as other low-level aspects such as covert channels. Nevertheless, as Proverif is sound, a formal correctness proof provided by Proverif guarantees the correctness of the analyzed model, which excludes logical weaknesses in the protocol, such as the ones illustrated in this paper. To address potential false negatives related to cryptography and other low-level aspects that cannot be represented in a Proverif symbolic model, a complementary analysis using a (finer) computational model would be required alongside the Proverif verification. In any case, it is important to note that implementation errors may introduce divergences from the analyzed models and they are another possible cause of vulnerabilities. In addition to false negatives, a formal verification may lead to false positives. As Proverif is sound but not complete, when it identifies an attack, the attack may not be possible in the model. While this problem can be solved by carefully analyzing the found attacks, and excluding the ones that cannot be reproduced in the model, an attack that is feasible in the model may not be feasible in a real implementation of the protocol, since the implementation may include additional low-level security checks that invalidate the attack in reality, resulting in false positives. For this reason, when Proverif returns an attack that is feasible on the model, it is still necessary to test that the attack is also feasible against a concrete implementation of the protocol, as we did.

In our work on the FDO protocol, we used formal verification several times: (i) to find the attacks on the FDO specification; (ii) to validate the countermeasures proposed by FIDO and demonstrate that they do not completely solve the security issue; (iii) to test the validity of our proposed solution to ensure its logical correctness.

In the next part of the section, we discuss the FDO models used for formal verification with Proverif. All the source. pv files used for verification are available in a dedicated repository at https://github.com/netgroup-polito/verification-fdo. The machine we used to run Proverif was equipped with an Intel i7-6700 CPU with 32 GB of RAM. All verification results were returned within 30 minutes after the start of the execution.

A. Proverif Component Model and Assumptions

The Proverif component model reflects the component model described in Section IV. Multiple devices can run the FDO protocol at the same time, but there is only one legitimate manufacturer, owner, and server, in addition to possibly other malicious ones. Each of these entities can participate in multiple

instances of the protocol with different devices. For example, the manufacturer can create several devices, the server can handle multiple connections, and the owner can onboard more than one device.

The OV is represented in the model as a certificate containing multiple entries in a chain. In principle, the OV can contain an unlimited number of entries. However, to reduce the complexity of the verification (and to allow Proverif to terminate in a reasonable time), we imposed a limit of three entries that can coexist in the OV simultaneously. This implies that a device can only change ownership up to three times before being onboarded on the Owner. We believe this assumption to be reasonable and not to curtail the security analysis conducted on the protocol. In fact, even with a chain of length $N = 3$ (e.g., Manufacturer $\rightarrow$ Reseller $\rightarrow$ Owner), Proverif is able to test all critical combinations of attacks, including: Replay Attack (use an old OV), Insertion of Malicious Intermediate Owner (e.g., Manufacturer $\rightarrow$ Attacker $\rightarrow$ Owner), Extension of the OV to multiple entities (e.g., Manufacturer $\rightarrow$ Attacker $\rightarrow$ Owner/Attacker). A longer OV chain does not introduce new categories of ownership transfers and already captures all relevant transitions (initial owner, intermediate entity, and final owner).

B. Proverif Channel and Attacker Model

The communications occurring in the various phases of the protocol were modeled in accordance with Section IV. In particular, we used public channels in Proverif to represent communications over public networks, and a private channel for the DI phase, which is assumed to take place in a secure environment.

Concerning the attacker, it is modelled as a classic Dolev-Yao attacker: it can intercept, read, modify, and replay the messages that travel on a public channel. It may represent a dishonest manufacturer, owner, server, a counterfeit device, or a combination of them. The attacker can even perform cryptographic operations but only using the keys it knows. On the other hand, as a modelling assumption, the attacker cannot perform side channel-attacks, nor brute force operations (e.g., it cannot revert a hash function nor decrypt a message without knowing the decryption key).

C. Security Properties

We used Proverif to assess the feasibility of the threats identified in the threat model section and summarised in Table II. With the exception of DoS, which cannot be modelled in Proverif, all the threats were modelled and analysed through symbolic queries. In particular, we used:

- Correspondence assertions to verify authentication and message integrity properties, which implies resistance to impersonation and traffic manipulation attacks;
- Injective correspondence assertions to verify absence of replay attacks;
- Reachability queries to verify confidentiality properties, i.e., the undisclosure of credentials and sensitive messages.

The attacks discussed in this paper correspond to violations of confidentiality properties (concerning the messages exchanged after the TO2 phase and the symmetric session key) and of authentication properties, capturing impersonation attacks in the different phases of the protocol. Table III summarises the verification results for these properties across the different protocol versions.

TABLE III
VERIFICATION RESULTS ON SECURITY PROPERTIES

Confidentiality	FDO protocol [8]	FIDO counterm. [10]	Our solution
After TO2 phase	✗(*1)	✗(*1)	✓

Authentication		FDO protocol [8]	FIDO counterm. [10]	Our solution
TO0 phase	Owner→Server	✗(*2)	✓	✓
	Server→Owner	✗(Not implemented by design, *3)	✗(Not implemented by design, *3)	✓
TO1 phase	Device→Server	✓	✓	✓
	Server→Device	✗(Not implemented by design, *4)	✗(Not implemented by design, *4)	✓
TO2 phase	Device→Owner	✓	✓	✓
	Owner→Device	✗(*5)	✗(*5)	✓

(*1) The attacker knowing the session key can read session messages
(*2) The attacker can store its entry on the Server
(*3) The attacker can execute TO0 with the Owner
(*4) The attacker can send its entry to the Device
(*5) The attacker can onboard the Device

IX. CONSIDERATIONS ON THE PROPOSED SOLUTION

Below are several final considerations related to the proposed solution, beginning with its advantages:

- The integration of a server certificate and the inclusion of its hash in the OVHeader robustly fortify the authentication process. This adjustment, together with the modifications applied across the protocol phases, mitigates all the attacks described in Section V.
- Using certificates for authentication independently of HTTPS enhances the protocol's adaptability, which is particularly beneficial in environments where HTTPS may not be feasible.
- Although the proposed changes require an additional signature verification on the device side (to authenticate the server), they do not introduce new cryptographic capabilities for the device. Signature verification is already required in the original protocol to authenticate the Owner during the TO2 phase.

The following aspects warrant further consideration:

- The reliance on a CA to issue server certificates may introduce an additional point of failure and increase the administrative overhead associated with certificate lifecycle management (e.g., certificate renewal and rotation).
- A server certificate revocation process must be considered for cases in which the server private key is compromised. This would require updating all OVs in which the compromised certificate is used.
- The management and validation of certificates in a large-scale deployment might require significant infrastructure support to ensure efficiency and scalability. This also needs to account for scenarios involving multiple rendezvous servers or infrastructure migration.

In conclusion, while the proposed modifications significantly enhance the security of the FDO protocol, they also introduce

operational challenges that may affect its adoption and scalability. These factors should be carefully evaluated to determine the most effective deployment strategy.

X. Conclusions

In this paper, we have analysed the security of the FDO protocol proposed by the FIDO Alliance and candidate to become a reference standard for automatic device onboarding. In a previous conference work, we had guessed a specific attack that can be conducted against the protocol when the supply chain of the device cannot be assumed as trusted and when the device cannot communicate over HTTPS (e.g., constrained device) [9]. In this article, after analysing the potential consequences of this vulnerability in detail, we successfully replicated the attack against an actual implementation of the protocol, proving that the previously identified issue is indeed exploitable. Then, we focused on possible countermeasures to mitigate the encountered security issue, and we analysed the mitigation strategies proposed by FIDO, showing that they are still vulnerable to MitM attacks when HTTPS is not used for device to Rendezvous Server communication. Finally, we proposed a variation on the protocol, which provides authentication of the Rendezvous Server in phases TO0 and TO1, thus mitigating all the attacks. It is important to emphasise that the proposed solution does not increase the computational and storage capabilities required for the device, and that its logical correctness has been proven using formal verification and Proverif.

Before submission, this paper was shared with the FIDO Alliance, which confirmed the validity of our findings. They also assured us that our proposed solution would be brought to the attention of the FDO Working Group for further discussion regarding its feasibility and impact following the new proposed features. Depending on what is decided by the Working Group, one might consider implementing our proposed solution as future work.

References

[1] "Internet of Things - Worldwide statistics," 2023. Accessed: Dec. 12, 2023. [Online]. Available: https://www.statista.com/outlook/tmo/internet-of-things/worldwide

[2] Y. Yang, L. Wu, G. Yin, L. Li, and H. Zhao, "A survey on security and privacy issues in Internet-of-Things," *IEEE Internet Things J.*, vol. 4, no. 5, pp. 1250–1258, Oct. 2017.

[3] "Industrial Internet of Things security framework (IISF)," 2023. Accessed: Dec. 12, 2023 [Online]. Available: https://www.iiconsortium.org/IISF/

[4] "IIC automated onboarding and device provisioning: Best practices," 2022. [Online]. Available: https://www.iiconsortium.org/iic-automated-onboarding-paper-form/

[5] "NIST project: Trusted Internet of Things device network-layer onboarding and lifecycle management," 2025. Accessed: Dec. 12, 2023. [Online]. Available: https://www.nccoe.nist.gov/projects/trusted-iot-device-network-layer-onboarding-and-lifecycle-management

[6] "Intel secure device onboarding (SDO)," 2025. [Online]. Available: https://www.intel.com/content/www/us/en/internet-of-things/secure-device-onboard.html

[7] G. Cooper et al., "Fido device onboard specification 1.0," *Rev. Draft*, Dec. 2020.

[8] G. Cooper et al., "Fido device onboard specification 1.1," *Rev. Draft*, Dec. 2021.

[9] S. Bussa, R. Sisto, and F. Valenza, "Formal verification of the FDO protocol," in *Proc. IEEE Conf. Standards Commun. Netw.*, 2023, pp. 290–295.

[10] "FDO ownership voucher double extension vulnerability guidance," 2023. [Online]. Available: http://fidoalliance.org/specs/fidoiot/appnote1-ov-guidance-v1.0-fd-20230306.html

[11] M. Ammar, G. Russello, and B. Crispo, "Internet of Things: A survey on the security of IoT frameworks," *J. Inf. Secur. Appl.*, vol. 38, pp. 8–27, 2018.

[12] P. Aufner, "The IoT security gap: A look down into the valley between threat models and their implementation," *Int. J. Inf. Secur.*, vol. 19, no. 1, pp. 3–14, 2020.

[13] "FDO security requirements," 2022. [Online]. Available: https://fidoalliance.org/specs/fdo-security-requirements/FDO-security-requirements-v1.0-fd-20220913.html

[14] "Index of /specs/fidoiot," 2025. [Online]. Available: http://fidoalliance.org/specs/fidoiot/

[15] B. Blanchet, B. Smyth, V. Cheval, and M. Sylvestre, "ProVerif 2.00: Automatic cryptographic protocol verifier, user manual and tutorial," Version from, vol. 1, pp. 05–16, 2018.

[16] K. Reaz and G. Wunder, "ASOP: A sovereign and secure device onboarding protocol for cloud-based IoT services," in *Proc. 6th Cyber Secur. Netw. Conf.*, 2022, pp. 1–3.

[17] R. Schermann, S. Bussa, R. Urian, and C. Steger, "Zero-touch privacy preserving provisioning in an Edge-, Fog-, and Cloud environment," in *Proc. Eighth Int. Conf. Fog Mobile Edge Comput.*, 2023, pp. 276–283.

[18] "FDO implementation source code," 2023. Accessed: Jan. 03, 2023. [Online]. Available: https://github.com/fido-device-onboard/pri-fidoiot



Simone Bussa received the MSc (*summa cum laude*) degree in computer engineering from the Politecnico di Torino, Turin, Italy, in 2021, where he is currently working toward the PhD degree in control and computer engineering. His research interests include distributed systems security and formal verification applied in the field of cyber-physical systems.



Riccardo Sisto (Member, IEEE) received the PhD degree in computer engineering from Politecnico di Torino, Italy, in 1992. Since 2004, he has been a full professor of computer engineering with Politecnico di Torino. He has authored or coauthored more than 100 scientific papers. His research interests include the area of formal methods, applied to distributed software and communication protocol engineering, distributed systems, and computer security. He is a senior member of ACM.



Fulvio Valenza (Member, IEEE) received the MSc (*summa cum laude*) degree and the PhD (*summa cum laude*) degree in computer engineering from the Politecnico di Torino, Torino, Italy, in 2013 and 2017, respectively. He is currently a tenure-track assistant professor with Politecnico di Torino. His research focuses on network security policies, orchestration and management of network security functions in SDN/NFV-based networks, and threat modeling.

Open Access funding provided by 'Politecnico di Torino' within the CRUI CARE Agreement