An Extensive Formal Analysis of Multi-factor Authentication Protocols

Charlie Jacomme, Steve Kremer

An extensive formal analysis of multi-factor authentication protocols

CHARLIE JACOMME, LSV, CNRS, ENS Paris-Saclay, Inria, and Université Paris-Saclay
STEVE KREMER, LORIA, Inria Nancy-Grand Est, CNRS, and Université de Lorraine

Passwords are still the most widespread means for authenticating users, even though they have been shown to create huge security problems. This motivated the use of additional authentication mechanisms in so-called multi-factor authentication protocols. In this paper we define a detailed threat model for this kind of protocols: while in classical protocol analysis attackers control the communication network, we take into account that many communications are performed over TLS channels, that computers may be infected by different kinds of malwares, that attackers could perform phishing, and that humans may omit some actions. We formalize this model in the applied pi calculus and perform an extensive analysis and comparison of several widely used protocols — variants of *Google 2-step* and *FIDO's U2F* (Yubico's Security Key token). The analysis is completely automated, generating systematically all combinations of threat scenarios for each of the protocols and using the Proverif tool for automated protocol analysis. To validate our model and attacks we demonstrate their feasibility in practice, even though our experiments are run in laboratory environment. Our analysis highlights weaknesses and strengths of the different protocols. It allows us to suggest several small modifications of the existing protocols which are easy to implement, as well as an extension of *Google 2-step*, that improves security in several threat scenarios.

CCS Concepts: • **Security and privacy → Formal security models**; **Multi-factor authentication**.

Additional Key Words and Phrases: formal methods; multi-factor authentication; detailed threat models; symbolic model

ACM Reference Format:
Charlie Jacomme and Steve Kremer. 2020. An extensive formal analysis of multi-factor authentication protocols. *ACM Trans. Priv. Sec.* 1, 1, Article 1 (January 2020), 35 pages. https://doi.org/10.1145/3440712

1 INTRODUCTION

Users need to authenticate to an increasing number of electronic services in everyday life: email and bank accounts, agendas, e-commerce sites, etc. Authentication generally requires a user to present an *authenticator*, that is *"something the claimant possesses and controls (typically a cryptographic module or password) that is used to authenticate the claimant's identity"* [14]. Authenticators are often classified according to their *authentication factor*:

- what you know, e.g., a password, or a pin code;
- what you have, e.g., an access card or physical token;
- what you are, e.g., a biometric measurement.

Although these different mechanisms exist, passwords are still by far the most widely used mechanism, despite the fact that many problems with passwords were already identified in the late '70s when they were mainly used to grant login into a computer [19]. Since then, things have become worse: many people choose the same weak passwords for many

Manuscript submitted to ACM

purposes, and large password databases have been leaked. Studies have shown that the requirement to add special characters does not solve these problems, and the latest recommendations by NIST [13] even discourage this practice.

To palliate password weaknesses, multi-factor authentication protocols combine several authentication factors. Typically, instead of using only a login and password, the user proves possession of an additional device, such as their mobile phone or a dedicated authentication token. Two popular protocols are *Google 2-step* [12] (which actually regroups several mechanisms) and *FIDO's U2F* [24] (the version implemented by Yubico for their Security Keys), which is supported by many websites, including Google, Facebook, and GitHub. In (one version of) *Google 2-step*, the user receives a verification code on their phone to be copied onto their computer, while *FIDO's U2F* requires the use of a specific USB token that must be plugged into the computer.

Our contributions. In classical protocol analysis, the attacker is supposed to control the communication network. However, the protocols we study in this paper make extensive use of TLS communications and are supposed to provide security even if some devices are infected by malware.

We therefore propose a novel, detailed threat model for multi-factor authentication protocols which takes into account many additional threats.

- Compromised passwords: our basic assumption is that the user's password has been compromised. Otherwise multi-factor authentication would not be required.
- Network control: we define a *high-level model of TLS channels* that guarantees confidentiality of messages, authentication of the server if the user verifies the corresponding certificate, and additionally ensures, through inclusion of session ids, that messages of different TLS sessions cannot be mixed. Nevertheless, we allow the attacker to delay or block messages. Our model also contains a notion of *fingerprint* that is used in some protocols to identify machines, and we may give the adversary the power to spoof such fingerprints.
- Compromised platforms: we give a structured and fine-grained *model for malwares*. We take an abstract view of a system as a set of input and output interfaces, on which an adversary may have read or write access, depending on the particular malware.
- Human aspects: we take into account that most of these protocols require some interaction with the *human user*. We model that humans may not correctly perform these steps. Moreover, we model that a human may be a victim of *phishing*, or *pharming*, and hence willing to connect to and enter their credentials on a malicious website.
- *"Trust this computer mechanism"*: to increase usability, several websites, including Google and Facebook, offer the possibility to *trust* a given machine, so that the use of a second factor becomes unnecessary on these machines. We add this trust mechanism to our model.

We completely formalize these threat scenarios in the applied pi calculus.

We analyse several variants of the *Google 2-step* and *FIDO's U2F* protocols in this model. The analysis is completely automated, using scripts to generate systematically all combinations of threat scenarios for each of the protocols and using the Proverif tool for automated protocol analysis. The scripts and Proverif source files are available at [22]. Even though we eliminate threat scenarios as soon as results are implied by weaker scenarios, the analysis required over 6 000 calls to Proverif, yet finishes in only a few minutes. Our analysis results in a detailed comparison of the protocols which highlights their respective weaknesses and strengths. It allows us to suggest several small modifications of the existing protocols which are easy to implement, yet improve their security in several threat scenarios. In particular, the existing mechanisms do not authenticate the *action* that is performed, e.g., a simple login may be substituted by a login enabling the *"trust this computer"* mechanism, or a password reset. Adding some additional information to the display

may thwart such attacks in many of our threat scenarios. We also propose a new variant of *Google 2-step* building on ideas from the *FIDO's U2F* protocol.

To validate our model and analysis we verify that the weaknesses we found can indeed be put into practice. We report on our experiments with the google mail server and a FIDO USB token, implementing the *FIDO's U2F* protocol. Even though our experiments are performed in a laboratory environment they confirm the relevance of our models and analyses.

Related work. Bonneau et al. [8] propose a detailed framework to classify and compare web authentication protocols. They use it for an extensive analysis and compare many solutions for authentication. While the scope of their work is much broader, taking into account more protocols, as well as usability issues, our security analysis of a more specific set of protocols is more fine-grained in terms of malware and corruption scenarios. Moreover, our security analysis is grounded in a formal model using automated analysis techniques.

Some other attempts to automatically analyse multi-factor authentication protocols were made, including for instance the analysis of *FIDO's U2F* [20], the *Yubikey One Time Password* [16, 17] and the *Secure Call Authorization* protocols [3]. However, those analyses do not study resistance to malware, nor do they capture precisely TLS channel behavior or fingerprints. Basin et al. [4] studied how human errors could decrease security. Their model is more evolved than ours on this aspect. However, we consider more elaborate malwares and also check for a stronger authentication property: an attack where both a honest user and an attacker try to log into the honest user's account but only the attacker succeeds is not captured in [4], as they simply check that every successful login was proceeded by an attempt from the corresponding user to login. In the same vein, [5] studies minimal topologies to establish secure channels between humans and servers. Their goal is to establish a secure channel, while we consider entity authentication. They consider authentic and confidential channels, which we extend by being more fine grained.

2 MULTI-FACTOR AUTHENTICATION PROTOCOLS

In this section we briefly present the two, widely used, multi-factor authentication protocols that we study in this paper: (several variants of) *Google 2-step* and *FIDO's U2F*.

2.1 *Google 2-step*

To improve security of user logins, Google proposes a two factor authentication mechanism called *Google 2-step* [12]. If enabled, a user may use his phone to confirm the login. On their website Google recalls several reasons why password-only authentication is not sufficient and states that *"2-Step Verification can help keep bad guys out, even if they have your password"*. *Google 2-step* proposes several variants. The default mechanism sends to the user, by SMS, a verification code to be entered into his computer. An alternative is the "One-Tap" version, where the user simply presses a Yes button in a pop-up on his phone. The second version avoids to copy a code and is expected to improve the usability of the mechanism. This raises an interesting question about the trade-off between security and ease of use. We also present a more recent version of "One-Tap" that we dubbed "Double-Tap".

As Google does not provide any detailed specification of the different authentication mechanisms, the following presentations are based on reverse engineering. As the protocols are simple and do not contain complex cryptographic operations, the reverse engineering is rather straightforward, based on the operations visible by the user and behavioral tests. Notice though that we may have omitted some checks performed by the server, based on some information that is not entered by the user, such as the timing of the login. As we validated inside a laboratory environment the attacks

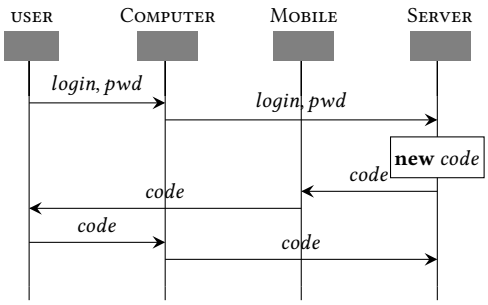

Fig. 1. G2V protocol

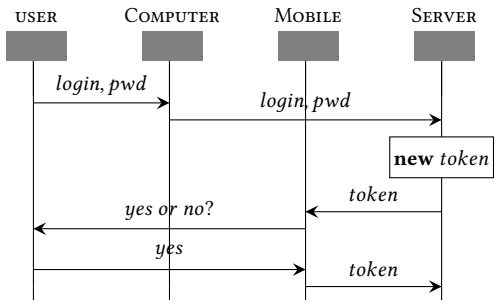

Fig. 2. G2OT protocol

found systematically, our protocol models appear to be precise enough. All experiences presented in this paper were performed in January and February 2018.

2.1.1 Google 2-step with verification codes - G2V. In Figure 1 we depict the different steps of the protocol. All communications between the user's computer and the server are protected by TLS. The three main steps of the protocol are:

(1) the user enters their login and password into their computer, which forwards the information to the server;
(2) upon receiving login and password, the server checks them. In case of success, the server generates a fresh 6 digits code, and sends an SMS of the form *"G-****** is your Google verification code"* to the user's mobile phone;
(3) the user then copies the code to their computer, which sends it to the server. If the correct code is received login is granted.

When the password is compromised, the security of the protocol only relies on the code sent on the SMS channel. Thus, if the attacker can intercept the code produced in step (2) before it is received by the server, the attacker could use the code to validate their own session and break the security. This could be done for instance by intercepting the SMS, compromising the phone with a malware, or through a key-logger on the user's computer.

2.1.2 Google 2-step with One-Tap - G2OT. In Figure 2 we present the One-Tap version of *Google 2-step*, the main steps being:

(1) the user enters their login and password into their computer, which forwards the information to the server;
(2) the server then creates a fresh random *token* that is sent to the user's mobile phone. Unlike in the previous version, the communication between the server and the phone is over a TLS channel rather than by SMS;
(3) the phone displays a pop-up to the user who can then confirm the action or abort it, by choosing "Yes" or "No" respectively;
(4) in case of confirmation the phone returns the token and login is granted.

Note that in its most basic version, the user only answers a yes/no question. Google announced in February 2017 [23] that the pop-up would also contain in the future a fingerprint of the computer, including information such as IP address, location and computer model. However this new version has yet to be implemented on some of the smartphones we used for tests. In the following we will analyse both versions, with (G2OT$^{\text{fpr}}$) and without (G2OT) the fingerprint. Remark that in steps (2) to (4), the authentication token is never sent to the computer. This is an important difference with the previous version, disabling attacks based on compromising the computer, e.g. with a key-logger. The independence

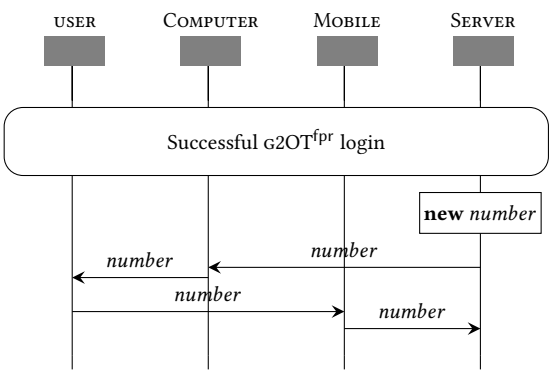

Fig. 3. G2DT^{fpr} protocol

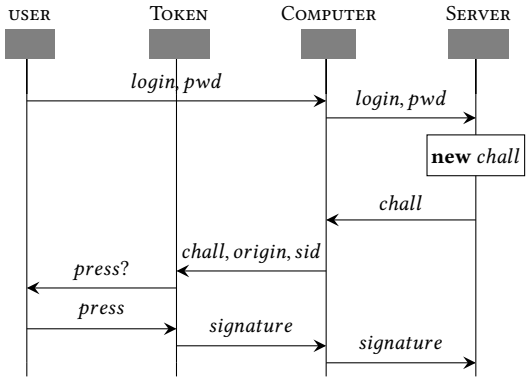

Fig. 4. U2F protocol

of the second factor with respect to the computer then improves the security. Adding a fingerprint to the screen additionally improves the security, as it allows the user to detect a suspicious login from an unknown location.

2.1.3 Google 2-step with Double-Tap - G2DT^{fpr}. The issue with One-Tap compared to the code version is that the user is likely to simply press "Yes" without reading any displayed information. To mitigate this issue, Google sometimes uses a version which we call Double-Tap. We were not able to find a public documentation of this variant, but we saw it at work in practice. The first step is the One-Tap protocol previously presented, with the display of the fingerprint. It is then followed by a second step, where a two digit number is displayed on the user's computer screen, and the same number is displayed on the user phone along with two other random numbers. The user is then asked to select on their phone the number displayed on their computer. This selection mechanism mimics the behavior of a verification code displayed on the computer and that the user should enter on their phone, but with the benefits of greater simplicity and ease of use. If we abstract the selection mechanism used to simplify the user experience and simply consider that the user is entering the data on their phone, the protocol outline is shown in Figure 3.

2.2 *FIDO's Universal 2nd Factor* - U2F

FIDO is an alliance which aims at providing standards for secure authentication. They propose many solutions under the U2F, FIDO and FIDO2 [6, 11] (also known as the WebAuthn) standards. We only study partially the Universal 2nd Factor (U2F) protocol [24], focusing on the version using a USB token as the second factor. More precisely, we study the implementation of the standard performed by the Yubico company, producing the Yubikey token. The U2F protocol relies on a token able to securely generate and store secret and public keys, and perform cryptographic operations using these keys. Moreover, the token has a button that a user must press to confirm a transaction. To enable second-factor authentication for a website, the token generates a key pair[1] and the public key is registered on the server. This operation is similar to the registration of a phone as a second factor in the case of a Google account. We must assume that this step was performed securely by the user at a time where their password was secure, and ideally at the time of the creation of the account. Else, no security may come from the second factor. Once the registration has been performed, the token can then be used for authenticating; the steps of the authentication protocol are presented in Figure 4, and can be explained as:

[1]In the case of the Yubikey token, the key is generated by hashing a fresh random with a fixed secret stored inside the token.

(1) the computer forwards the user's login and password to the server;
(2) the server generates a challenge which is sent to the user's computer;
(3) upon reception, the browser generates a payload containing the URL of the server, the challenge and the identifier of the current TLS session to be signed by the token;
(4) the user confirms the transaction by pressing the token button;
(5) the token signs the payload, and the signature is forwarded to the server for verification.

Compared to G2OT and G2DT^{fpr}, the second factor and the user's computer are not independent, which may lead to attacks base on malware on the computer. However, thanks to the signature of the payload, the signature sent back to the server is strongly linked to the current session, and session confusion is significantly harder. Moreover, as the signature includes the URL seen by the user, this may counter phishing attacks.

2.3 Disabling the second factor on trusted devices

When designing an authentication protocol, as also emphasized in [8], a key requirement should be usability. On a user's main computer, used on a daily basis, it may not be necessary to use a second factor: for instance, using a second factor each time a user pops their emails on their main laptop would be very cumbersome. This may explain why several providers, including Google and Facebook, propose to *trust* specific computers and disable the second factor authentication on these particular machines. This is done by checking a *"Trust this computer"* option when initiating a two-factor authenticated login on a given machine. Technically, the computer will be identified by a cookie and its *fingerprint*. A fingerprint typically includes information about the user's IP address, inferred location, OS or browser version, etc. As those elements will obviously change over time, in practice, a distance between fingerprints is evaluated, and if the fingerprint is too far from the expected one, the second factor authentication will be required. To the best of our knowledge, this feature is not documented and the full mechanism has not been studied previously even though it may lead to security issues. To capture such security issues we will include the *"Trust this computer"* mechanism in our analysis.

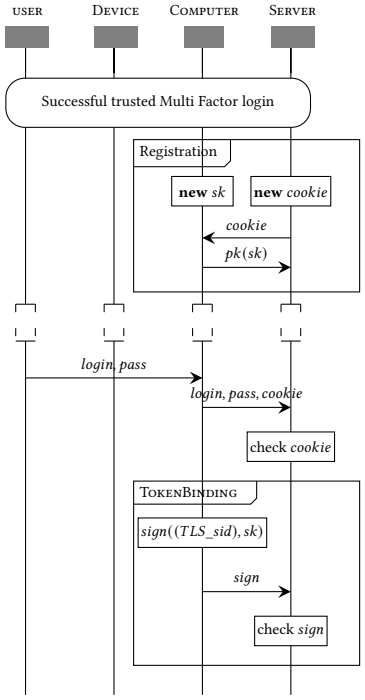

Fig. 5. TOKENBINDING

2.4 Token Binding

While cookies are a common mechanism widely used to remember a computer after a successful login, a new protocol called TOKENBINDING [21] is under development. Its usage is recommended by the FIDO standards, but providers are free to use it or not. After a successful login, a public key may be bound to the user account, and the corresponding secret key will be used to sign the session identifier of the following TLS sessions. It may be seen as a partial U2F where the keys are directly stored on the computer. We describe the protocol in Figure 5. If a computer has been successfully authenticated, the registration part of TOKENBINDING may be enabled and the computer may generate a new secret key, and simply send the corresponding public key to the server.

In parallel, the server may send a classical cookie to the computer. For later logins, the server will ask for the cookie but also for the signature of the TLS session identifier by the registered public key. We remark that the cookie and the signature may actually be sent at the same time, and TokenBinding thus does not require more communications than classical cookie authentication after the registration.

3 THREAT MODEL

In order to conduct an in depth analysis of multi-factor authentication protocols, we consider different threat models, types of attacks and corresponding attacker capabilities. We will consider a Dolev-Yao attacker [9] that controls any compromised parts and, classically, the network. However, many of the protocols we study use channels protected by TLS. The attacker may block a message, even if he cannot read or write on such channels. Moreover, as we are studying multi-factor authentication protocols, in order to assess additional protection offered by these protocols, we are interested in the case where the user's password has been compromised. Therefore, the most basic threat scenario we consider is the one where the attacker has (partial) control over the network, and knows the users' passwords.

There are however several ways the attacker can gain more power. Our aim is to present a detailed threat model, reflecting different attacker levels that may have more or less control over the user's computer, the network, or even over the user itself. Those levels aim at capturing the attacker capabilities that are necessary for a given attack.

3.1 Malware based scenarios

The first range of scenarios covers malwares that give an attacker control over parts of a user's device, also known as Man In The Machine attacks.

3.1.1 Systems as interfaces. To give a principled model of malwares and what parts of a system the malware may control, we take an abstract view of a system as a set of *interfaces* on which the system receives inputs and sends outputs. Some interfaces may only be used for inputs, while other interfaces may be used for outputs, or both. For example the keyboard is an input interface, the display is an output interface, and the network is an input and output interface. Compromise of part of the system can then be formalized by giving an attacker read or write access to a given interface. On a secure system, the attacker has neither read nor write access on any interface. Conversely, on a fully compromised system the attacker has read-write access on all interfaces.

More formally we consider that for each interface the attacker may have no access (NA), read-only access (RO), write-only access (WO), or read-write access (RW).

We may specify many different levels of malware by specifying for every interface two access levels, one for inputs and one for outputs on the interface. Obviously, for a given interface not all combinations need to be considered: a read-write access will yield a stronger threat model than read-only access, write-only or no access.

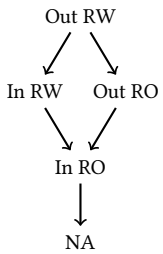

Fig. 6. Access lattice

We will suppose in this paper that it is harder to control the outputs of an interface than its inputs: therefore a given access level to the outputs will imply the same access level on the interface inputs. Although not a limitation of our model, this choice is motivated by practical considerations. Running for instance a key-logger does not require specific rights, because the keyboard data is completely unprotected in the OS. FIDO devices are identified by the OS as a keyboard (at least on Linux systems). However, reading data sent by an application to a USB device, i.e., having read access on the USB interface's output, may require to corrupt the driver (or in the case of Linux

enable the "USBmon" module) which requires specific privileges. Similarly, we suppose that having write access implies having read access. This yields for each interface five levels, that can be organised as a lattice depicted in Figure 6.

3.1.2 Malware on a computer. For a computer, we will consider four interfaces:

- the USB interface, capturing for instance the keyboard, or a U2F USB key, with all possible types of access;
- the display, the computer screen, with only output interfaces;
- the TLS interface, capturing the network communications, but by always assuming that the attacker has the same level of control over inputs and outputs;
- the hard drive interface, capturing control of the storage of the computer, with all possible types of access.

We can succinctly describe a malware on a computer by giving for each interface the attacker's rights for both inputs and outputs of this interface. We use the notation $\mathcal{M}^{\mathsf{interf}}_{\mathsf{in}:acc1,\mathsf{out}:acc2}$, where interf might be TLS, USB, hdd or dis, and $acc1$ and $acc2$ might be $\mathcal{RO}$ or $\mathcal{RW}$, to denote that the attacker has rights $acc1$ on the inputs, respectively rights $acc2$ on the outputs, of interface interf.

By convention, if we do not specify any access level, it means that the attacker has no access. A key-logger is for instance denoted with $\mathcal{M}^{\mathsf{USB}}_{\mathsf{in}:\mathcal{RO}}$. If the access level is the same both for the inputs and the outputs, as we always assume for TLS, we may write $\mathcal{M}^{\mathsf{TLS}}_{\mathsf{io}:\mathcal{RW}}$, thus capturing the fact that the attacker may have full control over the user browser, or that he might have exploited a TLS vulnerability.

Remark that we give a very high-level threat model for TLS, only considering read and write accesses. While this subsumes all possible capabilities, this does not reflect precisely the capabilities that an attacker may gain through XSS or CSRF attacks. However, such attacks tend to be linked to the actual implementation of the webserver or of the browser, rather than being protocol specific. Furthermore, capturing those attacks requires a very fine grained model of the web infrastructure, similar to the one presented by Fett et al. [10]. Such a fine grained model would break the automation of our analysis, which was already at the limit of Proverif's capabilities (minor changes to the model lead to non termination of Proverif).

3.1.3 Malware on a phone. For a mobile phone, the type of interface may depend on the protocols, with for instance SMS inputs or TLS inputs. To simplify, we will consider a phone to have only one input and one output interface. We thus only consider a generic device interface called dev, with all possible access levels. $\mathcal{M}^{\mathsf{dev}}_{\mathsf{in}:\mathcal{RO}}$ then corresponds for instance to the attacker having broken the SMS encryption, or to some malware on the phone listening to inputs.

3.2 Fingerprint Spoofing

Whenever a user browses the Internet, the user provides information about him or herself, called their *fingerprint*. Those elements will be very useful later on for additional checks in our protocols, and as we mentioned Google is adding this kind of details to their One-Tap protocol. However, in some cases the attacker might be able to obtain the same fingerprint as a given user. While some elements, such as the OS version, are rather easy to spoof, it is more complicated to spoof the IP address and inferred location. It is nevertheless possible if an attacker either completely controls the network the user connects on, or is connected to the same WiFi, or works in the same office.

3.3 Human errors

The attacker may also exploit vulnerabilities that rely on the user not or wrongly performing some actions, or preferring to ignore security warnings. The assumption that users may not behave in the expected way seems reasonable given that

most users are not trained in computer security, and their goal is generally to access a service rather than performing security related actions.

3.3.1 Phishing. In our model, we capture that users may be victims of *phishing* attempts, i.e. willing to authenticate on a malicious website. For instance, an untrained, naive user may be willing to click on a link in an email which redirects to a fake web site. While a phishing attack through an e-mail may not fool a trained user, even a more experienced user may be victim to more sophisticated attacks, for instance if he connects to an attacker WiFi hotspot which asks to login to a website in order to obtain free WiFi. Therefore, when we consider the *phishing threat scenario* we allow the attacker to choose with whom the user will initiate the protocol. We consider phishing as one of the simplest attacks to mount, and protocols should effectively protect users against it.

However, even though we consider that users might be victim of phishing, we suppose that they are careful enough to avoid it when performing the most sensitive operations: these operations include the registration of the U2F key, and logging for the first time on a computer they wish to trust later on. Indeed, if we were to allow phishing to be performed during those steps, no security guarantees could ever be achieved as the use of a second factor authentication requires a trusted setup.

3.3.2 No compare. A protocol may submit to the user a fingerprint and expect the user to continue the protocol only if the fingerprint corresponds to their own. When given a fingerprint and a confirmation button, some users may confirm without reading the displayed information. Thus, when considering the *no compare* scenario, we assume that the user does not compare any given value and always answers yes.

3.4 Threat scenarios considered

In our analysis we consider all the possible combinations of the previously presented scenarios. This yields a fine-grained threat model that allows for a detailed comparison of the different protocols, and to identify the strengths and weaknesses of each protocol, by showing which threats are mitigated by which mechanisms.

By considering those possibilities, we capture many real life scenarios. For instance, when a user connects to a WiFi hotspot in a hotel or train station, the WiFi might be controlled by the attacker, making the fingerprint spoofing and phishing scenarios realistic, because the attacker can have full control over the network, and thus use the provided IP address or redirect a user to a fake website.

If we try to connect on some untrusted computer, for instance the computer of a coworker, it may contain a rather basic malware, for instance a key-logger ($\mathcal{M}^{\mathsf{USB}}_{\mathsf{in}:\mathcal{RO}}$). However, if we connect on a computer shared by many people at some place, for instance at a cybercafe, there could be a very strong malware controlling the display of the computer ($\mathcal{M}^{\mathsf{dis}}_{\mathsf{out}:\mathcal{RW}}$) or controlling any TLS connection on this computer ($\mathcal{M}^{\mathsf{TLS}}_{\mathsf{io}:\mathcal{RW}}$). Moreover, the network in this unknown place might also be compromised, and we may have some other scenarios combined with the malware, such as phishing (PH) or fingerprint spoofing (FS).

Our different scenarios provide different levels of granularity going from no attacker power at all to complete control over both the network and the platform. Our threat model abstracts away from how the attacker gained this power. Thus, the scenarios we consider will contain at some point all the possible attacks, without the need to specify how they may be performed. Note that we distinguish access to the RAM of the computer and access to the hard drive. For instance, a TLS session key will only be stored in RAM and a cookie will be stored on the hard drive. A side channel attack such as Meltdown [18] or Spectre [15] may allow the attacker to read the RAM of the user computer. In the protocols studied in this paper, all values stored inside the RAM are received over one of the channels and not generated

by the computer. Thus, in our examples the RAM read only access is equivalent to giving read-only access to all the interfaces of the computer ($\mathcal{M}^{\mathsf{USB}}_{\mathsf{io}:\mathcal{RO}}\ \mathcal{M}^{\mathsf{TLS}}_{\mathsf{io}:\mathcal{RO}}\ \mathcal{M}^{\mathsf{dis}}_{\mathsf{io}:\mathcal{RO}}\ \mathcal{M}^{\mathsf{hdd}}_{\mathsf{io}:\mathcal{RO}}$). Another threat scenario is pharming, where the attacker can "lie" about the URL that is displayed to the user. This may happen either because of a malware that edits the *hosts* file (on a UNIX system), or by performing DNS ID Spoofing or DNS Cache Poisoning. All of these scenarios are simply captured as $\mathcal{M}^{\mathsf{TLS}}_{\mathsf{io}:\mathcal{RW}}$.

4 THE FORMAL MODEL

For our formal analysis, we model protocols in a dialect of the applied pi calculus [1, 2] that serves as input language to the Proverif tool [7] which we use to automate the analysis. We will only give a brief, informal overview here, which should be sufficient to explain our modelling of TLS sessions and threat scenarios. We refer the reader to [7] for additional details about the formal semantics.

4.1 The applied-pi calculus and Proverif

In the applied pi-calculus, protocols are modelled as concurrent processes that communicate messages built from a typed term algebra. The attacker controls the execution of the processes and can make computations over known terms. The grammar is given in Figure 7.

Atomic terms are either names $a, b, c, n, \ldots$, or variables $x, y, z, \ldots$, each declared with a type. Pre-defined types include channel, boolean and bitstring, but a user may define additional types. We note that the type system is only a convenient way to avoid errors in the specification; it does not limit the attacker, and types are basically ignored in the semantics. We suppose a set of function symbols, split into *constructors* and *destructors*. Each function symbol has an *arity*, defining the number and types of the arguments, as well as the type of the resulting term. Terms are built by applying constructors to other terms. Destructors are defined by one or several rewriting rules and model the properties of function symbols. For example, we can model digital signatures as follows. Suppose that `pkey` and `skey` are user defined types, modelling public and secret keys. Then we can define the function constructors

$$\mathsf{pk}(\mathtt{skey}) : \mathtt{pkey} \text{ and } \mathtt{sign}(\mathtt{bitstring}, \mathtt{skey}) : \mathtt{bitstring}$$

as well as the destructor `checksign` by the following rewrite rule

$$\mathtt{checksign}(\mathtt{sign}(m, k), \mathsf{pk}(k)) \rightarrow m$$

While constructors are used to build terms, application of destructors generalizes terms to expressions. Expressions may *fail* when a destructor is applied and the expression cannot reduce to a term by applying the rewrite rules defining the destructors. Additionally, one may declare *equations* on terms, which define a congruence relation on terms that are considered equal. Hence, an alternative way of specifying digital signatures would be to declare `checksign` as a constructor together with the equation

$$\mathtt{checksign}(\mathtt{sign}(m, k), \mathsf{pk}(k)) = m$$

In contrast to the previous modelling, $\mathtt{checksign}(t_1, t_2)$ is a valid term for any t_1, t_2 and the evaluation of this term will not fail. Moreover, one can define *private* names and function symbols that may not be used by the attacker.

The protocols themselves are modelled by processes. $\mathbf{0}$ is the terminal process that does nothing. $P \mid Q$ runs processes P and Q in parallel, and $!P$ allows to spawn an unbounded number of copies of P to be run in parallel. $\mathbf{new}\ n : T$ declares a fresh name of type T; this construct is useful to generate fresh, secret keys and nonces. $\mathbf{in}(M, x : T)$ inputs a

$M, N ::=$	terms	$P, Q ::= \mathbf{0}$	null process
$\mid a, b, c, k, m, n, s$	name	$\mid P \parallel Q$	parallel
$\mid x, y, z$	variable	$\mid !P$	replication
$\mid \mathsf{f}(M_1, \ldots, M_n)$	constructor application	$\mid \mathbf{new}\ n : T.\ P$	name restriction
		$\mid \mathbf{in}(M, x : T).P$	message input
$E ::=$	expressions	$\mid \mathbf{in}(M, = N).P$	pattern matching
$\mid M$	name	$\mid \mathbf{out}(M, N).P$	message output
$\mid \mathsf{h}(E_1, \ldots, E_n)$	function application	$\mid \mathbf{if}\ E_1 = E_2\ \mathbf{then}\ P\ \mathbf{else}\ Q$	conditional
$\mid \mathsf{fail}$	failure	$\mid \mathbf{event}\ e(M).P$	event e

Fig. 7. Terms and processes

term that will be bound to a variable of type T on channel M, $\mathbf{in}(M, = N)$ will only continue if the term N is provided on channel M; $\mathbf{out}(M, N)$ outputs the term N on channel M. If the channel name is known to (or can be computed by) the adversary, the channel is under the adversary's control: any input message may come from the adversary, and any output is added to the adversary's knowledge. On the contrary, if the channel is private, then the adversary can neither read from nor write to this channel. The conditional $\mathbf{if}\ E_1 = E_2\ \mathbf{then}\ P\ \mathbf{else}\ Q$ checks whether two expressions successfully evaluate to equal terms and executes P, or Q if at least one of the expressions failed or the two expressions yield different terms. Finally processes can be annotated by an event $e(M)$ where e is a user defined event. Events do not influence the process execution and serve merely as an annotation for specifying properties.

Throughout the paper we will use some of the usual syntactic sugar and simplifications: we generally omit the final $\mathbf{0}$ process, as well as $\mathbf{else}\ 0$ branches, and as in ProVerif we write $\mathbf{in}(c, = t).P$ instead of $\mathbf{in}(c, x).\mathbf{if}\ x = t\ \mathbf{then}\ P\ \mathbf{else}\ 0$.

As an example consider the processes defined in Figure 8. A user process $User$ wants to authenticate to some server $Server$. To do so, the user sends their $login$ and $password$ to their platform which are then forwarded to the server. The $Server$ generates a fresh $code$ sent to the user's $Mobile$. The code is then forwarded to the user, and back to the server through the platform.

In this paper we are interested in verifying *authentication properties*. We model them, as usual, as correspondence properties of the form

$$e_1(t_1, \ldots, t_n) \Longrightarrow e_2(u_1, \ldots, u_m)$$

Such a property holds, if in each execution, every occurrence of an instance of $e_1(t_1, \ldots, t_n)$ is preceded by the corresponding instance of $e_2(u_1, \ldots, u_m)$. Considering the example of Figure 8, we model the property that any accepted login was actually initiated by the user

$$Login(login) \Longrightarrow Initiate(login)$$

This property is satisfied here, thanks to the sms channel which is private. An even stronger property is verified, as each $Login$ can be matched with exactly one $Initiate$. For such properties, we use *injective* correspondence properties

$$e_1(t_1, \ldots, t_n) \Longrightarrow_{\mathsf{inj}} e_2(u_1, \ldots, u_m)$$

that require that each occurrence of e_1 is matched by a *different* preceding occurrence of e_2.

channel a.
channel sms [private].
channel kb [private].
channel $phone$ [private].

$Server(login : \texttt{bitstring}, passwd : \texttt{bitstring}) \hat{=}$
  in(a, x);
  if $x = (login, passwd)$ **then**
    new $code : \texttt{bitstring}$;
    out$(sms, code)$;
    in$(a, = code)$;
    event Login$(login)$.

$Platform \hat{=}$
  in$(kb, (x_{login} : \texttt{bitstring}, x_{passwd} : \texttt{bitstring}))$;
  out$(a, (x_{login}, x_{passwd})$;
  in$(kb, x_{code} : \texttt{bitstring})$;
  out(a, x_{code}).

$User(login : \texttt{bitstring}, passwd : \texttt{bitstring}) \hat{=}$
  event Initiate$(login)$;
  out$(kb, (login, passwd))$;
  in$(phone, x_{code} : \texttt{bitstring})$;
  out(kb, x_{code}).

$Mobile \hat{=}$
  in$(sms, x_{code} : \texttt{bitstring})$;
  out$(phone, x_{code})$.

new $login : \texttt{bitstring}$; **new** $passwd : \texttt{bitstring}$; $!Server(login, passwd)|!Platform|!Mobile|!User(login, passwd))$

Fig. 8. *Google 2-step* toy example

4.2 Modelling TLS communications

Most web protocols rely on TLS to ensure the secrecy of the data exchanged between a client and a server. In order to formally analyse online authentication protocols, we thus need to model TLS sessions and corresponding attacker capabilities. A possibility would of course be to precisely model the actual TLS protocol and use this model in our protocol analysis. This would however yield an extremely complex model, which would be difficult to analyse. A more detailed model of TLS would mostly be of interest for the analysis of TLS itself, rather than the protocol that make use of it. Therefore, for this paper, we opt to model TLS at a higher level of abstraction.

In essence we model that TLS provides

- confidentiality of the communications between the client and the server, unless one of them has been compromised by the adversary;
- a session identifier that links all messages of a given session, avoiding mixing messages between different sessions.

We model this in the applied pi calculus as follows:

- we define a private function TLS(id, id) : channel where id is a user defined type of identities, and use the channel TLS(c, s) for communications between client c and server s;
- we define a *TLS manager* process that given as inputs two identities id_1 and id_2 outputs on a public channel the channel name TLS(id_1, id_2), if either id_1 or id_2 are compromised;
- we generate a fresh name of type sid for each TLS connection and use it as a session identifier, concatenating it to each message, and checking equality of this identifier at each reception in a same session.

$Platform(idP) \triangleq$
  $\mathbf{in}(kb, (x_{login}, x_{passwd}, x_{ids}))$;
  $\mathbf{out}(\mathtt{TLS}(idP, ids), (x_{login}, x_{passwd}))$;
$\|$ $\mathbf{in}(kb, x_{code})$;
  $\mathbf{out}(\mathtt{TLS}(idP, ids), x_{code})$.

$TLS\_manager(idP, idS) \triangleq$
  $\mathbf{in}(a, (idc, ids))$;
  if $\mathtt{not}(idc = idP) \| \mathtt{not}(ids = idS)$ **then**
    $\mathbf{out}(a, \mathtt{TLS}(idc, ids))$.

$Server(login, passwd, idS) \triangleq$
  $\mathbf{in}(a, x_{client})$;
  $\mathbf{in}(\mathtt{TLS}(x_{client}, idS), (= login, = passwd))$;
  new $code$;
  $\mathbf{out}(sms, code)$;
  $\mathbf{in}(\mathtt{TLS}(x_{client}, idS), = code)$;
  event $\mathtt{Login}(login)$.

$User(login, passwd, idS) \triangleq$
  event $\mathtt{Initiate}(login)$;
  $\mathbf{out}(kb, (login, passwd, idS))$;
  $\mathbf{in}(phone, x_{code})$;
  $\mathbf{out}(kb, x_{code})$.

new $login$; **new** $passwd$; **new** idS; **new** idP;
  $!Server(login, passwd, idS) | !Platform(idP) | !TLS\_manager(idS, idP) | !Mobile | !User(login, passwd))$

Fig. 9. *Google 2-step* toy example with TLS

However, even if the communication is protected by TLS, we suppose that the adversary can block or delay communications. As communications over private channels are synchronous we rewrite each process of the form $\mathbf{out}(\mathtt{TLS}(c, s), M).P$ into a process $\mathbf{out}(\mathtt{TLS}(c, s), M)|P$. This ensures that the communications on TLS channels are indeed *asynchronous*. We provide the new elements of our previous toy example in Figure 9 (we omit the types for concision).

The TLS manager essentially allows the attacker to have a valid TLS session as long as the communication is not between the honest user and the server. This means that, even though we consider a single honest user, the attacker can perform all actions corresponding to sessions involving other users. Hence, in our model we consider a single honest user in parallel with an arbitrary number of corrupted users. As the corrupted user may behave honestly, considering a single honest user is not a limitation. Note however that we assume that there are no interactions between the user's computer and phone and the equipment of other users.

4.3 Modelling threat models

We will now present how we model the different scenarios discussed in Section 3 in the applied pi calculus.

4.3.1 Malware. As discussed in Section 3.1.1, we view a system as a set of interfaces. By default, these interfaces are defined as private channels. Let a be a public channel. A malware providing read-only access to an interface ch is modelled by rewriting processes of the form $\mathbf{in}(ch, x).P$ into processes of the form $\mathbf{in}(ch, x).\mathbf{out}(a, x).P$, respectively $\mathbf{out}(ch, M).P$ into $\mathbf{out}(a, M).\mathbf{out}(ch, M).P$, depending on whether inputs or outputs are compromised. Read-write access is simply modelled by revealing the channel name ch, which gives full control over this channel to the adversary. We provide in Figure 10 an example where the input received on the keyboard channel kb is forwarded to the attacker. The modified part of the process is highlighted.

4.3.2 Fingerprint and spoofing. As discussed before, when browsing, one may extract information about a user's location, computer, browser and OS version, etc. This fingerprint may be used as an additional factor for identification, and can also be transmitted to a user for verification of its accuracy. We model this fingerprint by adding a function

```
channel kb [private].
Platform(idP : id)≙
  in(kb, (login, passwd, ids));
  out(a, (login, passwd, ids));
```

Fig. 10. Key-logger example

```
User(login, passwd, idP, idS)≙
  event Initiate(login);
  out(a, (login, passwd, idS));
  in(phone, (code, fingerprint));
  if fingerprint = fpr(idP) then
    out(a, code).
```

Fig. 11. Fingerprint example

```
User(login, passwd, idP, idS)≙
  event Initiate(login);
  in(a, ida : id);
  if ida = idS then
    out(kb, (login, passwd, ida));
    in(phone, (code, fingerprint));
    if fingerprint = fpr(idP) then
      out(kb, code).
```

Fig. 12. Phishing example

$\mathtt{fpr}(id) : \mathtt{fingerprint}$ which takes an identity and returns its corresponding fingerprint. Given that all network communications are performed over a TLS channel $\mathtt{TLS}(c, s)$ the server s can simply extract the fingerprint $\mathtt{fpr}(c)$. However, in some cases we want to give the attacker the possibility to spoof the fingerprint, e.g., if the attacker controls the user's local network. In these cases we declare an additional function $\mathtt{spoof}_{\mathtt{fpr}}(\mathtt{fingerprint}) : \mathtt{id}$ and the equation

$$\mathtt{fpr}(\mathtt{spoof}_{\mathtt{fpr}}(\mathtt{fpr}(c))) = \mathtt{fpr}(c)$$

which provides the attacker with an identity whose fingerprint is identical to $\mathtt{fpr}(c)$, and allows the attacker to initiate a communication on a channel $\mathtt{TLS}(\mathtt{spoof}_{\mathtt{fpr}}(\mathtt{fpr}(c)), s)$.

We show in Figure 11 an example where the *User* also receives from their phone the *fingerprint* of the platform seen by the server, and checks that the fingerprint does match the fingerprint of their platform.

4.3.3 Human errors - No compare. Our model contains dedicated processes that represent the expected actions of a human, e.g., initiating a login by typing on the keyboard, or copying a received code through the display interface of their computer or phone. A user is also assumed to perform checks, such as verifying the correctness of a fingerprint or comparing two random values, one displayed on the computer and one on the phone. In the *No Compare* scenario we suppose that a human does not perform these checks and simply remove them. The corresponding process is obtained from Figure 11, by simply removing the highlighted conditional "**if** *fingerprint* = $\mathtt{fpr}(idP)$ **then** ".

4.3.4 Human errors - Phishing. In our model of TLS we simply represent a URL by the server identity *idS*, provided by the human user, as it was shown in Figure 9. This initiates a communication between the user's computer, with identifier *idC*, and the server over the channel $\mathtt{TLS}(idC, idS)$. This models that the server URL is provided by the user and may be the one of a malicious server, which their machine is then connecting to. We let the adversary provide the server identity *idA* to the user in order to model a basic *phishing* mechanism. We distinguish two cases: a trained user will check that $idA = idS$, where *idS* is the correct server, while an untrained user will omit this check and connect to the malicious server. The updated *User* process is provided in Figure 12, where we highlight the line to be removed under phishing.

5 ANALYSIS AND COMPARISON

In this section we use the formal framework to analyse several multi-factor authentication protocols. The analysis is completely automated using the Proverif tool. All scripts and source files used for these analyses are available at [22].

5.1 Properties and methodology

5.1.1 Properties. We focus on authentication properties and consider that a user may perform 3 different actions:

- an *untrusted login*: the user performs a login on an untrusted computer, i.e., without selecting the *"trust this computer"* option, using second-factor authentication;
- a *trusted login*: the user performs an initial login on a trusted computer, and selects the *"trust this computer"* option, using second-factor authentication;
- a *cookie login*: the user performs a login on a previously trusted computer, using their password but no second factor, and identifying through a cookie and fingerprint.

For each of these actions we check that whenever a login happens, the corresponding login was requested by the user. We therefore define three pairs of events

$$(init_x(\mathtt{id}), accept_x(\mathtt{id})) \qquad x \in \{u, t, c\}$$

The $init_x(\mathtt{id})$ events are added to the process modelling the human user, in order to capture the user's intention to perform the login action. The $accept_x(\mathtt{id})$ events are added to the server process. The three properties are then modelled as three injective correspondence properties:

$$accept_x(id) \Longrightarrow_{\mathsf{inj}} init_x(id) \qquad x \in \{u, t, c\}$$

When the three properties hold, we have that every login of some kind accepted by the server for a given computer matches exactly one login of the same kind initiated by the user on the same computer.

5.1.2 Methodology. For every protocol, we model the three different types of login, and then check using Proverif whether each security property holds for all possible (combinations of) threat scenarios presented in Section 3. As we consider trusted and untrusted login, we provide the user with two platforms: a trusted platform on which the user will try to perform trusted logins, and an untrusted platform for untrusted logins. We will thus extend the notation for malwares presented in 3.1.2 by prefixing the interface with t if the interface belongs to the trusted computer, and u if it belongs to the untrusted computer. For instance, $\mathcal{M}^{\mathsf{u-usb}}_{\mathsf{in}:\mathcal{RO}}$ corresponds to a key-logger on the untrusted computer. A scenario is described by a list of considered threats that may contain

- phishing (PH);
- fingerprint spoofing (FS);
- no comparisons by the user (NC);
- the malwares that may be present on the trusted and untrusted platform.

For instance, "PH FS $\mathcal{M}^{\mathsf{t-usb}}_{\mathsf{io}:\mathcal{RW}}$" denotes the scenario where the attacker can perform phishing, fingerprint spoofing, and has read-write access to the inputs and outputs of USB devices of the trusted computer. "NC $\mathcal{M}^{\mathsf{u-tls}}_{\mathsf{io}:\mathcal{RW}}$ $\mathcal{M}^{\mathsf{u-usb}}_{\mathsf{io}:\mathcal{RW}}$ $\mathcal{M}^{\mathsf{u-dis}}_{\mathsf{io}:\mathcal{RW}}$" models a human that does not perform comparisons and an attacker that has read-write access to the inputs and outputs of the TLS, USB and display interfaces of the untrusted device.

We use a script to generate the files corresponding to all scenarios for each protocol and launch the Proverif tool on the generated files. In total we generated 6 172 scenarios that are analysed by Proverif in 8 minutes on a computing server with twelve Intel(R) Xeon(R) CPU X5650 @ 2.67GHz and 50Go of RAM. We note that we do not generate threat scenarios whenever properties are already falsified for a weaker attacker (considering less threats or weaker malware). The script generates automatically the result tables, displaying only results for minimal threat scenarios that provide

Threat Scenarios		G2V	G2OT	G2OT^{fpr}
		✓	✗	✓
PH		✗	✗	✓
NC		✓	✗	✗
FS		✓✓✓	✗	✗
	$\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✓✓✓	✗	✓
	$\mathcal{M}^{dev}_{in:\mathcal{RO}}$	✗	✗	✓
	$\mathcal{M}^{t-dis}_{io:\mathcal{RO}}$	✓	✗	✓
	$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✗	✗	✓
	$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$	✗	✗	✓
	$\mathcal{M}^{dev}_{in:\mathcal{RW}}$	✗	✗	✗
	$\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✗	✗	✗✓✗
FS	$\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✓✓✗	✗	✗
	$\mathcal{M}^{u-hdd}_{in:\mathcal{RO}}$	✓	✗	✓
	$\mathcal{M}^{u-dis}_{io:\mathcal{RO}}$	✓	✗	✓
	$\mathcal{M}^{u-tls}_{io:\mathcal{RO}}$	✗	✗	✓
	$\mathcal{M}^{u-usb}_{in:\mathcal{RO}}$	✗	✗	✓
	$\mathcal{M}^{u-tls}_{io:\mathcal{RW}}$	✗	✗	✓✗✗
	$\mathcal{M}^{u-usb}_{in:\mathcal{RW}}$	✗	✗	✓✗✓

Table 1. Analysis of the basic *Google 2-step* protocols

Threat Scenarios		G2DT^{fpr}
NC		✓
FS		✓
NC	$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓
NC	$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$	✓
FS	$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$	✓
NC	$\mathcal{M}^{u-tls}_{io:\mathcal{RO}}$	✓
NC	$\mathcal{M}^{u-usb}_{in:\mathcal{RO}}$	✓
FS	$\mathcal{M}^{u-tls}_{io:\mathcal{RO}}$	✓
FS	$\mathcal{M}^{u-usb}_{in:\mathcal{RO}}$	✓

Table 2. Analysis of the *Google 2-step* Double-Tap

attacks, and maximal threat scenarios for which properties are guaranteed. In the following sections we present partial tables with results for particular protocols. Full results for all protocols are given in Tables 9 and 10 in Appendix.

The result tables use the following notations:

- results are displayed as a triple $u\ t\ c$ where u, t, c are each ✗ (violated) or ✓ (satisfied) for the given threat scenario; each letter in the set $\{u, t, c\}$ gives the status of the authentication property for *u*ntrusted login, *t*rusted login and *c*ookie login respectively;
- ✖ and ✔ are shortcuts for ✗✗✗ and ✓✓✓;
- signs are greyed when they are implied by other results, i.e., the attack existed for a weaker threat model, or the property is satisfied for a stronger adversary;
- we sometimes use blue, circled symbols to emphasize differences when comparing protocols.

Even if Proverif can sometimes return false attacks, we remark that any ✗ corresponds to an actual attack where Proverif was able to reconstruct the attack trace.

5.2 *Google 2-step*: Verification Code and One-Tap

In this section we report on the analysis of the currently available *Google 2-step* protocols: the verification code (G2V, described in Section 2.1.1), the One-Tap (G2OT, described in Section 2.1.2) with and without fingerprint, and the Double-Tap (G2DT^{fpr}, described in Section 2.1.3). The results are summarized in Tables 1 and 2.

5.2.1 G2V. In the G2V protocol the user must copy a code received on their phone to their computer to validate the login. We first show that G2V is indeed secure when only the password of the user was revealed to the attacker: as long as the attacker cannot obtain the code, the protocol remains secure. If the attacker obtains the code, either using a key-logger ($\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$), or by reading the SMS interface ($\mathcal{M}^{dev}_{in:\mathcal{RO}}$), or any other read access to an interface on which

the code is transmitted, the attacker can use this code to validate their own session. Looking at Table 1, it may seem surprising that a malware on a trusted platform may compromise an untrusted login. This is due to the fact that a code of a trusted session may be used to validate an untrusted session and vice-versa. Moreover, if the attacker can access the hard disk drive ($\mathcal{M}^{\mathsf{t-hdd}}_{\mathsf{in}:ro}$) he may steal the cookie that allows to login without a second factor, and then perform a login if he can also spoof the platform fingerprint (FS).

We have tested on the Google website that a code generated for a login request can indeed be used (once) for any other login, demonstrating that such attacks are indeed feasible. Interestingly, this also shows that in the actual implementation, the verification code is not linked to the TLS session. Not linking codes to sessions is actually useful as it allows to print in advance a set of codes, e.g., if no SMS access is available. Moreover, we note that linking the code to a session does actually not improve security in our model, as the code of the attacker session will also be sent to the user's phone and could then be recovered. In practice, if the code is linked, an attack can be produced only if the attacker's code is received first, i.e., if the attacker can login just before or after the user.

We remark that the results for G2V are also valid for another protocol, *Google Authenticator*. On this protocol the phone and the server share a secret key, and use it to derive a one time password (OTP) from the current time. In all the scenarios where the SMS channel is secure, G2V can be seen as a modelling of *Google Authenticator* where the OTP is a random value "magically" shared by the phone and the server.

5.2.2 G2OT. In the G2OT protocol a user simply confirms the login by pressing a yes/no button on their phone. We first consider the version that does not display the fingerprint, and which is still in use. Our automated analysis reports a vulnerability even if only the password has been stolen. In this protocol, the client is informed when a second, concurrent login is requested and the client aborts. However, if the attacker can block, or delay network messages, a race condition can be exploited to have the client tap yes and confirm the attacker's login. We have been able to reproduce this attack in practice and describe it in more detail in Section 6. While the attack is in our most basic threat model, it nevertheless requires that the attacker can detect a login attempt from the user, and can block network messages (as supposed in the Dolev-Yao model).

5.2.3 G2OT$^{\mathsf{fpr}}$. We provide in the third column of Table 1 the analysis of G2OT$^{\mathsf{fpr}}$. To highlight the benefits of the fingerprint, we color additionally satisfied properties in blue. In many read only scenarios ($\mathcal{M}^{\mathsf{t-tls}}_{\mathsf{io}:\mathcal{RO}}$, $\mathcal{M}^{\mathsf{t-usb}}_{\mathsf{in}:\mathcal{RO}}$,$\mathcal{M}^{\mathsf{u-tls}}_{\mathsf{io}:\mathcal{RO}}$, $\mathcal{M}^{\mathsf{u-usb}}_{\mathsf{in}:\mathcal{RO}}$), and even in case of a phishing attempt, the user sees the attacker's fingerprint on their phone and does not confirm. However, if the user does not check the values (NC) or if the attacker can spoof the fingerprint (FS), G2OT$^{\mathsf{fpr}}$ simply degrades to G2OT and becomes insecure. Some attacks may be performed on the cookie login, for instance for scenarios $\mathcal{M}^{\mathsf{t-tls}}_{\mathsf{io}:\mathcal{RW}}$ or $\mathcal{M}^{\mathsf{t-usb}}_{\mathsf{io}:\mathcal{RW}}$, as the attacker may initiate a login from the user's computer without the user having any knowledge of it, and then use it as a kind of proxy.

Because of the verification code, in scenarios FS or NC, G2V provides better guarantees than G2OT$^{\mathsf{fpr}}$. It is however interesting to note that G2OT$^{\mathsf{fpr}}$ resists to read only access on the device as there is no code to be leaked to the attacker. One may argue that an SMS channel provides less confidentiality than a TLS channel, i.e., the read-access on the SMS channel may be easier to obtain in practice. Indeed, SMS communications between the cellphone and the relay can be made with weaker encryption (A5/0 and A5/2) than TLS, and the SMS message will anyway be sent over TLS between the relay and the provider's servers. While this argument is in favour of G2OT$^{\mathsf{fpr}}$, one may also argue that G2V has better resistance to user inattention, as a user needs to actively copy a code.

Threat Scenarios					G2V^{fpr}	G2V^{dis}	G2OT^{dis}	G2DT^{dis}
PH					✓	✓	✔	✔
PH	FS				✗	✗✓✓	✗	✗✓✓
PH	FS	$\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$			✗	✗✓✗	✗	✗✓✗
PH	FS	$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$			✗	✗	✗	✗✓-
PH	FS	$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$			✗	✗	✗	✗✓✓
PH	FS	$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$			✗	✗✓✓	✗	✗
PH	FS	$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$	$\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$		✗	✗	✗	✗✓✗
PH	FS	$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	$\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$		✗	✗✓✗	✗	✗
		$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$			✓	✓	✓✓✓	✓
		$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$			✓	✓	✔	✓
		$\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$			✗✓✗	✓✓✗	✓✓✗	✓✓✗
	FS	$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$			✗	✓✗✗	✗	✓✓-
	FS	$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$			✗	✓✗✗	✗	✓✓✓
	FS	$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$			✓✓✓	✓✓✓	✗	✓✗✗
	FS	$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$	$\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$		✗	✓✗✗	✗	✓✓✗
	FS	$\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$			✗	✓✗✗	✗	✓✗✗
	FS	$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$			✗	✓✗✗	✗	✓✗✗
	FS	$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	$\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$		✓✓✗	✓✓✗	✗	✓✗✗
	FS	$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$		✗	✓✗✗	✗	✓✗✗
	FS	$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$	$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$		✗	✓✗✗	✗	✓✗✗
		$\mathcal{M}^{u-tls}_{io:\mathcal{RO}}$			✓	✓	✓✓✓	✓
		$\mathcal{M}^{u-usb}_{in:\mathcal{RO}}$			✓	✓	✓✓✓	✓
		$\mathcal{M}^{u-tls}_{io:\mathcal{RW}}$			✓✗✗	✓	✓✓✓	✓✓✓
		$\mathcal{M}^{u-usb}_{in:\mathcal{RW}}$			✓✗✓	✓	✓✓✓	✓✓✓
	FS	$\mathcal{M}^{u-tls}_{io:\mathcal{RO}}$			✗	✗✓✓	✗	✔
	FS	$\mathcal{M}^{u-usb}_{in:\mathcal{RO}}$			✗	✗✓✓	✗	✔
	FS	$\mathcal{M}^{u-dis}_{io:\mathcal{RW}}$			✔	✔	✗	✗✓✓
	FS	$\mathcal{M}^{u-tls}_{io:\mathcal{RW}}$			✗	✗✓✓	✗	✗✓✓
	FS	$\mathcal{M}^{u-usb}_{in:\mathcal{RW}}$			✗	✗✓✓	✗	✗✓✓
	FS	$\mathcal{M}^{u-dis}_{io:\mathcal{RW}}$	$\mathcal{M}^{u-tls}_{io:\mathcal{RO}}$		✗	✗✓✓	✗	✗✓✓
	FS	$\mathcal{M}^{u-usb}_{in:\mathcal{RO}}$	$\mathcal{M}^{u-dis}_{io:\mathcal{RW}}$		✗	✗✓✓	✗	✗✓✓

Table 3. *Google 2-step* protocols with additional display

5.2.4 G2DT^{fpr}. To palliate the weakness of G2OT compared to G2V, Google proposes G2DT^{fpr} where a comparison through a second tap is required. The additional security provided by the second tap is displayed in Table 2. Note that, as G2DT^{fpr} is strictly more secure than G2OT^{fpr}, we only report on the differences between them, that are highlighted in blue. The attacker must be able to have their code displayed and selected on the user's device in order to successfully login. Therefore, FS or NC scenarios with some additional read only access, are secure. Interestingly, in the NC scenario, we are now as secure as G2V, while having greater usability. We note that we are still not secure in the PH FS scenario. This means that an attacker controlling the user's network or some WiFi hotspot could mount an attack against G2DT^{fpr}.

5.3 Additional display

In this section, we propose and analyse small modifications of the previously presented protocols. Given the benefits discussed in section 5.2.3, we first add a fingerprint to G2V.

In *Google 2-step* some attacks occur because the attacker is able to replace a trusted login by an untrusted one, e.g. under $\mathcal{M}^{\mathsf{u-usb}}_{\mathsf{in}:\mathcal{R}'\mathcal{W}}$. If this happens, the attacker can obtain a session cookie for their own computer and perform additional undetected logins later on. A user might expect that by using a second factor, he should be able to securely login once on an untrusted computer and be assured that no additional login will be possible.

We now study a variant of each of the protocols where the user's action (trusted or untrusted login) is added to the display. This addition may create some harmless "attacks" where the attacker replaces a trusted login with an untrusted login. However, such attacks indicate that an attacker may change the type of action, such as password reset, or disabling second-factor authentication.

We call $\text{G2V}^{\mathsf{fpr}}$ the protocol version that additionally displays the fingerprint, and $\text{G2V}^{\mathsf{dis}}$, $\text{G2OT}^{\mathsf{dis}}$ and $\text{G2DT}^{\mathsf{dis}}$ the versions that additionally display fingerprint, respectively the action, and provide in Table 3 the results of our analysis. To highlight the benefits of our modifications, we color additionally satisfied properties in blue, when considering G2V and $\text{G2V}^{\mathsf{fpr}}$, $\text{G2V}^{\mathsf{fpr}}$ and $\text{G2V}^{\mathsf{dis}}$, $\text{G2OT}^{\mathsf{fpr}}$ and $\text{G2OT}^{\mathsf{dis}}$ and $\text{G2DT}^{\mathsf{fpr}}$ and $\text{G2DT}^{\mathsf{dis}}$.

It appears that adding the action - and the fingerprint in the G2V case - performs as expected: the protocols become secure in all the scenarios where the only possible attack was a mixing of actions.

5.4 Conclusion regarding *Google 2-step*

Currently, Google proposes G2V, G2OT, $\text{G2OT}^{\mathsf{fpr}}$ and $\text{G2DT}^{\mathsf{fpr}}$. Adding the type of login being performed (trusted or untrusted) to the display would provide additional security guarantees.

Among the studied mechanisms, $\text{G2V}^{\mathsf{dis}}$ and $\text{G2DT}^{\mathsf{dis}}$ provide the best security guarantees in our model, having each advantages and disadvantages. In Table 4, we provide a comparison between these two mechanisms. We observe that $\text{G2V}^{\mathsf{dis}}$ performs better than $\text{G2DT}^{\mathsf{dis}}$ only in scenarios where we have $\mathcal{M}^{\mathsf{t-dis}}_{\mathsf{io}:\mathcal{R}'\mathcal{W}'}$, which may be considered as a powerful malware.

$\text{G2DT}^{\mathsf{dis}}$ provides better guarantees in many simpler threat scenarios, with for instance read-only access to the phone. As the code is sent back to the server from the phone rather than the computer, this mechanism is more resilient to malware on the computer. Moreover, the code is sent through a TLS channel rather than via SMS, which may arguably provide better security.

Finally, even though *Google 2-step* may significantly improve security, phishing attacks combined with fingerprint spoofing are difficult to prevent. This seems to be inherent to the kind of protocol, where the security is only enforced through the 2nd factor. As we will see in the next section the FIDO U2F protocol may provide better guarantees for these threat scenarios.

5.5 FIDO U2F

FIDO's U2F adds cryptographic capabilities to the mechanism through its registration mechanism. As explained previously, the URL of the server the user is trying to authenticate to is included in the query made to the FIDO USB token, and also in the signature returned by the token. The server will then only grant login if the signature contains their own URL.

Threat Scenarios				G2V^{dis}	G2DT^{dis}
PH	FS		$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✗	✗✓-
PH	FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$	✗	✗✓✓
PH	FS		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✗✓✓	✗
PH	FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$ $\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✗	✗✓✗
PH	FS		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✗✓✗	✗
			$\mathcal{M}^{dev}_{in:\mathcal{RO}}$	✗	✓
		NC	$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✗	✓
		NC	$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$	✗	✓
		NC	$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✓	✗
	FS		$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✗✗	✓✓-
	FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$	✓✗✗	✓✓✓
	FS	NC	$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✗	✓✓-
	FS		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✓✓✓	✓✗✗
	FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$ $\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✓✗✗	✓✓✗
	FS	NC	$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$ $\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✗	✓✓✗
	FS		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✓✓✗	✓✗✗
	FS	NC	$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✓✓✗	✗
		NC	$\mathcal{M}^{u-tls}_{io:\mathcal{RO}}$	✗	✓
		NC	$\mathcal{M}^{u-usb}_{in:\mathcal{RO}}$	✗	✓
		NC	$\mathcal{M}^{u-dis}_{io:\mathcal{RW}}$	✓	✗
	FS		$\mathcal{M}^{u-tls}_{io:\mathcal{RO}}$	✗✓✓	✓✓✓
	FS		$\mathcal{M}^{u-usb}_{in:\mathcal{RO}}$	✗✓✓	✓✓✓
	FS		$\mathcal{M}^{u-dis}_{io:\mathcal{RW}}$	✓✓✓	✗✓✓

Table 4. Comparison of *Google 2-step* with code or tap

Threat Scenarios			U2F
			✓
PH			✓
	FS		✓✓✓
		$\mathcal{M}^{dev}_{in:\mathcal{RO}}$	- - -
		$\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✓✓✓
		$\mathcal{M}^{t-dis}_{io:\mathcal{RO}}$	✓
		$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✓✓
		$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$	✓
		$\mathcal{M}^{dev}_{io:\mathcal{RO}}$	✗
		$\mathcal{M}^{dev}_{in:\mathcal{RW}}$	✗
		$\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✗
		$\mathcal{M}^{t-usb}_{io:\mathcal{RW}}$	✗
	FS	$\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✓✓✗
	FS	$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✓✗
		$\mathcal{M}^{u-hdd}_{in:\mathcal{RO}}$	✓
		$\mathcal{M}^{u-dis}_{io:\mathcal{RO}}$	✓
		$\mathcal{M}^{u-tls}_{io:\mathcal{RO}}$	✓
		$\mathcal{M}^{u-usb}_{in:\mathcal{RO}}$	✓✓✓
		$\mathcal{M}^{u-tls}_{io:\mathcal{RW}}$	✗
		$\mathcal{M}^{u-usb}_{in:\mathcal{RW}}$	✓✗✓
		$\mathcal{M}^{u-usb}_{io:\mathcal{RW}}$	✗✗✗

Table 5. U2F results

We present the results of our formal analysis in Table 5. U2F is secure under many threat scenarios, including some that combine phishing and fingerprint spoofing. However, an attack is found when the computer runs malware that controls the USB interface of the trusted computer ($\mathcal{M}^{t-usb}_{io:\mathcal{RW}}$). Indeed, the malware can then communicate with the U2F token, and thus send a request generated for an attacker session. Also, if the attacker can control the TLS interface ($\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$ or $\mathcal{M}^{u-tls}_{io:\mathcal{RW}}$), he may change the intended action and replace an untrusted login with a trusted one. As a consequence, a login on an untrusted computer with U2F may enable future attacker logins on this computer. This contradicts claims that Yubikeys (an implementation of U2F token) guarantee protection against "phishing, session hijacking, man-in-the-middle, and malware attacks." While the claim indeed holds for the first threats, malware attacks are still possible. Moreover, one might expect an external hardware token to allow users to securely log on an untrusted computer. However, this enables an attacker to submit their own request to the user's token. Even though a user has to press the token button to accept each request, as noted previously, a malware controlling the TLS connection will allow several attacker logins for one user press due to the *"trust this computer"* mechanism.

U2F may lead to another problem that is out of the scope of our analysis: a Yubikey does not have any way to provide feedback for a successful press. When the computer submits two requests in a row to the token and the user just presses once, the user may believe that the press failed, and press once more. This is reminiscent of the problem identified during the analysis of the One-Tap mechanism: success and failure of the second factor should not be silent.

Threat Scenarios				U2F	$U2F_{tb}$	$G2DT^{dis}$	$G2DT^{dis}_{tb}$
PH	FS		$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✓✗	✓✓✓	✗✓-	✗✓✓
PH	FS	NC	$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✓✗	✓✓✓	✖	✖
PH	FS		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✓✗	✓✓✓	✖	✖
PH	FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✓✗	✓✓✓	✖	✖
	FS		$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✓✗	✓✓✓	✓✓-	✓✓✓
	FS		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✓✗	✓✓✓	✓✗✗	✓✗✗
	FS	NC	$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✓✗	✓✓✓	✖	✖
	FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✓✗	✓✓✓	✓✗✗	✓✗✗
	FS	NC	$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✓✗	✓✓✓	✖	✖

Table 6. Results for the TokenBinding extension

To summarize, one might expect U2F to protect against malware, as it is based on a secure hardware token providing cryptographic capabilities. Thus, even if U2F does provide a better security than most existing solutions, it does not uphold this promise completely. However, the U2F mechanism providing protection against phishing is very interesting. What appears to be lacking from U2F is some feedback capabilities, i.e a screen, to notify failures, successes, and maybe information such as the fingerprint of the computer.

5.6 Token Binding

We previously studied the security of the protocols combined with the *"trust this computer"* mechanism where a cookie is used to authenticate a computer on the long term. We provide in Table 6 the results of the formal analysis of TokenBinding combined with U2F and $G2DT^{dis}$, and highlight in blue the security gained with respect to the classical cookie version. It provides protection against a read only access to the TLS interface, because it is not any more sufficient to steal the cookie. We do not gain protection against control of the computer memory, as the secret key is stored the same way as the cookie. The attacker needs to be able to access the private key of the user which was generated on their platform but never sent over the network.

5.7 A $G2DT^{dis}$ extension : $G2DT^{ext}$

5.7.1 Core idea. We propose an extension of $G2DT^{dis}$ based on ideas from U2F. Our goal is to provide a protocol which will have the same user experience as $G2DT^{dis}$, but will provide a stronger protection against phishing by using the second factor to confirm the origin and the TLS session id.

To protect against phishing, the URL seen by the user must be authenticated. This requires an external intervention. For instance, in U2F the browser extracts the URL and sends it to the token. With a phone serving as a second factor, we may mimic this behavior by having the browser transmit the URL to the phone through some secure channel. The phone can then transmit the URL to the server on an independent channel, allowing the server to check that the URL seen by the user corresponds to their own URL. This however requires an efficient way to transmit data from the computer to the phone, ideally without any particular setup. NFC like technologies may provide a promising means for such a channel. Without this channel the selection mechanism of $G2DT^{dis}$ may be used: to verify that a user has seen some data on their computer the phone displays the data among other random data, and asks the user to select the correct one.

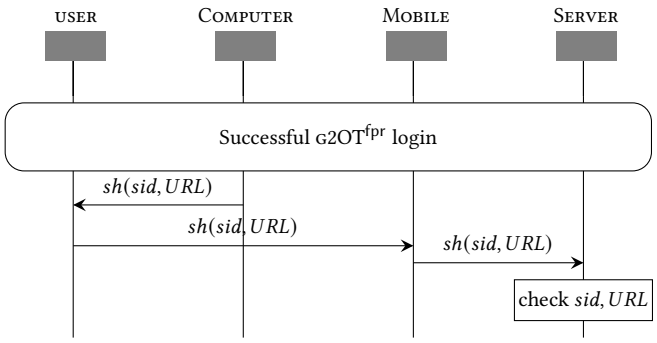

Fig. 13. g2DT^{ext} outline

Of course, an efficient and easy to deploy channel would be preferable to the selection mechanism. Yet, the selection mechanism of g2DT^{dis} allows for a protocol with the same user experience, rather easy to deploy, but with greater security.

5.7.2 Extension description. The extension is similar to g2DT^{dis}, except that the server does not send a freshly generated digit to the computer. The user's browser extracts the URL and the TLS session identifier and produces a short hash of those values that is displayed in a pop-up outside the web page. The server computes the same hash and sends it to the phone. The phone displays the hash among two other random values. If the user selects the correct value, the phone confirms the login to the server.

Intuitively, instead of using a signature to transmit securely the URL and the TLS session identifier, the protocol relies on a confirmation on the user's phone. The outline of the protocol is displayed in Figure 13.

Currently, g2DT^{dis} uses only a 2-digit integer. Hence, an attacker has probability 1/100 to guess the integer, which is much higher than usually accepted. If in g2DT^{ext} we were to use 2 digit hash values, an attacker could easily find collisions. To maintain usability and improve the security by transmitting more information, it might be worth exploring different mechanisms, such as using images or visual hashes. The only conditions are that the domain should be large, and a human should be able to instantly pick the correct value out of the three proposals.

5.8 g2DT^{ext} analysis

In Table 7 we provide the advantages of g2DT^{ext} when compared to g2DT^{dis}, where we highlight the newly secure cases in blue. We ensure higher security guarantees in some common scenarios such as phishing combined with a distracted user who does not compare values. This means that g2DT^{ext} can be used to effectively protect untrained people against phishing. Moreover, it is also secure in the case of phishing and fingerprint spoofing. Hence, the protocol provides secure login even when connecting on an untrusted network. The comparison between U2F and g2DT^{ext} is displayed in Table 8, where we highlight differences in blue. We do not display the device malware scenarios, that are not relevant for U2F, but in which case it naturally provides better security. To summarize, U2F is more secure against an attacker who can manipulate the display of the computer, or of course the phone itself. g2DT^{ext} is more secure against an attacker who can manipulate the USB ports of the computer or the network. It is difficult to say which protocol provides the best security as it depends on more practical considerations, that we discuss in the Section 7.1.

Threat Scenarios				G2DT^{dis}	G2DT^{ext}
PH		NC		✖	✓
PH	FS			✗✓✓	✓✓✓
PH	FS		$\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✗✓✗	✓✓-
PH	FS		$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✗✓-	✓✓✗
PH	FS	NC	$\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✖	✓✓-
PH	FS	NC	$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✓✓✗
PH	FS		$\mathcal{M}^{t-hdd}_{in:\mathcal{RW}}$	✗✓✗	✓✓✗
PH	FS		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✖	✓✗✗
PH	FS	NC	$\mathcal{M}^{t-hdd}_{in:\mathcal{RW}}$	✖	✓✓✗
PH	FS		$\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✖	✓✓✗
PH	FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$	✖	✓
PH	FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$ $\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✖	✓✓✗
PH	FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✓✓✗
		NC	$\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✗✗✗	✓✓✗
		NC	$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$	✖	✓
	FS		$\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✓✗✗	✓✓✗
	FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$	✓✗✗	✓✓✓
	FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$ $\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✓✗✗	✓✓✗
	FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✗✗	✓✓✗
	FS	NC	$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$ $\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✖	✓✓✗
	FS	NC	$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✓✓✗
		NC	$\mathcal{M}^{u-tls}_{io:\mathcal{RW}}$	✖	✓
		NC	$\mathcal{M}^{u-usb}_{in:\mathcal{RW}}$	✖	✓✗✓
		NC	$\mathcal{M}^{u-usb}_{in:\mathcal{RW}}$ $\mathcal{M}^{u-tls}_{io:\mathcal{RW}}$	✖	✓✗✗
	FS		$\mathcal{M}^{u-tls}_{io:\mathcal{RW}}$	✗✓✓	✓✓✓
	FS		$\mathcal{M}^{u-usb}_{in:\mathcal{RW}}$	✗✓✓	✓✓✓

Table 7. Comparison between G2DT^{dis} and G2DT^{ext}

Threat Scenarios			U2F	G2DT^{ext}
		$\mathcal{M}^{dev}_{in:\mathcal{RO}}$	- - -	✓
		$\mathcal{M}^{dev}_{io:\mathcal{RO}}$	✖	✓
		$\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✖	✓✓✗
	NC	$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✓	✖
		$\mathcal{M}^{t-usb}_{io:\mathcal{RW}}$	✖	✓
FS		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✓✓✓	✓✗✗
FS		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✓✓✗	✓✗✗
FS		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✓✗	✓✗✗
FS	NC	$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✓✓✗	✖
FS	NC	$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✓✗	✖
FS		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✖	✓✗✗
FS		$\mathcal{M}^{t-usb}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-hdd}_{in:\mathcal{RO}}$	✖	✓✓✗
FS		$\mathcal{M}^{t-usb}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✓✓✗
FS		$\mathcal{M}^{t-usb}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✖	✓✗✗
		$\mathcal{M}^{u-tls}_{io:\mathcal{RW}}$	✖	✓
	NC	$\mathcal{M}^{u-dis}_{io:\mathcal{RW}}$	✓	✖
		$\mathcal{M}^{u-usb}_{in:\mathcal{RW}}$	✓✗✓	✓✓✓
	NC	$\mathcal{M}^{u-usb}_{in:\mathcal{RW}}$ $\mathcal{M}^{u-dis}_{io:\mathcal{RW}}$	✓✗✓	✖
		$\mathcal{M}^{u-usb}_{io:\mathcal{RW}}$	✗✗✗	✓
	NC	$\mathcal{M}^{u-usb}_{in:\mathcal{RW}}$ $\mathcal{M}^{u-tls}_{io:\mathcal{RW}}$	✖	✓✗✗
	NC	$\mathcal{M}^{u-usb}_{io:\mathcal{RW}}$	✖	✓✗✓
FS		$\mathcal{M}^{u-dis}_{io:\mathcal{RW}}$	✓✓✓	✗✓✓
FS		$\mathcal{M}^{u-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{u-tls}_{io:\mathcal{RW}}$	✖	✗✓✓
FS		$\mathcal{M}^{u-usb}_{in:\mathcal{RW}}$ $\mathcal{M}^{u-dis}_{io:\mathcal{RW}}$	✓✗✓	✗✓✓
FS		$\mathcal{M}^{u-usb}_{io:\mathcal{RW}}$ $\mathcal{M}^{u-dis}_{io:\mathcal{RW}}$	✖	✗✓✓

Table 8. Comparison between U2F and G2DT^{ext}

6 VALIDATING ATTACKS IN PRACTICE

We provide below a more practical description of a few selected types of attacks and weaknesses. Demonstrating that these attacks can be put in practice, albeit in laboratory conditions, validates our protocol and attacker models. Some of these attacks were found with Proverif, while others were discovered during the reverse engineering of the protocols. We do not claim novelty of those attacks, which are not particularly complex. We provide for each type of attacks

- the outline of the required steps of the attacks,
- high level comments about the severity of the attacks and an understanding of how they work and why they are possible;
- a description of how the attack was validated inside a lab environment, that explains how exactly those attacks might be performed, by whom and what are the required capabilities.

Each attack was reproduced inside a laboratory setting with laptops and an internet connection, using a dedicated Google account for the G2OT attacks, and the first version of the "Security Keys series by Yubico" along with an open source API[2] for the FIDO attacks. Remark that some of the following weaknesses might also be combined into stronger attacks.

[2]https://github.com/Yubico/libu2f-server

6.1 Session confusion on G2V

Outline. The different steps of the attack are as follows:

(1) The user enters their email and password, initiating a user session;
(2) the browser informs the user that a code will be received on their phone;
(3) the attacker enters the user's login and password on another computer, initiating an attacker session;
(4) the attacker intercepts the code intended for the user session;
(5) the attacker uses the code of the user session to validate the attacker session.

Comments. The fact that the code generated to validate the user session can be used to validate the attacker session may be surprising. It implies that the attacker does not need to intercept the code intended for their own session, but can use the code of any user session. This is an important observation: if the attacker uses for instance a key-logger, the code that the user enters on their computer is the first one received, which is most likely the code for the first session, i.e., the user session. If the codes were linked to the sessions on the server side, the code entered on their computer by the user would be useless to the attacker. We also remark that, as previously mentioned, the SMS channel might not provide a high level of security, at least compared to TLS. Hence, it might be possible for an attacker to obtain the verification code through a weakness of the SMS channel. This weakness does not directly lead to a severe attack but it may facilitate performing some of the following attacks.

Attack Validation. We created a fresh Google account and enabled the second factor authentication by associating a previously unregistered phone. Using two distinct computers, we initiated a first login attempt on the first one and received a first code. We then initiated a second session on the second computer and received a second code. The second code was then used to validate the first session, and conversely. This confirms that the code sent is not linked to a specific login attempt.

6.2 Session confusion on G2OT

Outline. The different steps of the attack are as follows:

(1) the user enters their password and email, initiating a user session;
(2) the browser displays a request to confirm the request on their phone;
(3) the attacker detects that the user contacted the server. After the first reply from the server the attacker blocks all further messages;
(4) the attacker enters the user's login and password on another computer, initiating an attacker session;
(5) depending on the timing, two things may then happen:
    - the user presses yes, nothing happens on their screen, and the attacker is logged in;
    - or the user presses yes, nothing happens on their screen, but another yes/no pops up on their phone. If the user presses yes once more, the attacker is logged in.

Comments. A robust implementation should reject any kind of simultaneous login from different sessions, or at least display it clearly on the phone, as it is done in the browser. We believe it to be plausible that users, after having pressed yes on their phone without a successful login, would press yes a second time. This attacks relies inherently on a lack of feedback given to the user, and a lack of a strong link between the computer that starts the session and the phone

that validates it. This attack is concerning because of its simplicity. Google is implementing G2OT^{fpr}, but G2OT is still deployed on older mobile phones. It might be advisable to disable G2OT entirely.

Attack Validation. This attack was easy to reproduce in practice as it does not involve any complex manipulation. Using again a dedicated Google account, we

(1) initiated two sessions for the same user on two distinct computers,
(2) disconnected one computer from the network to reflect that the attacker blocks the network, and
(3) validated the session of the other computer on the phone.

Sometimes we had to confirm twice on the phone to validate the malicious session, and sometimes only once. In a basic version of this attack, which does not require to block the network, an attacker observing the target user could initiate a session just a few moments after the user, and be logged in when the target validates on their phone. The target would see an error of the type *"Something went wrong"* on their computer and might retry to login. However, the error message may appear before the user validates, as it appears as soon as a simultaneous login attempt is made. Thus, the more evolved version of this attack implies to block the Internet connection of the target computer after the user started their login. In our experiment, we simply temporarily disabled in step (2) the WiFi connection of the computer to disconnect the computer from the network. This effectively models an attacker that has the control over the network. Indeed, if the attacker controls the network, the attacker can detect all connections to the server IP address, and block all but the first connections through a firewall rule (although we did not implement the experiment detecting automatically the connection).

6.3 Phishing attack on *Google 2-step*

Outline. The different steps of the attack are as follows:

(1) the attacker directs the user to some malicious web page;
(2) the attacker initiates a login attempt with the server, and the malicious web page simply forwards every information and query from the login attempt to the user through the malicious web page.

Comments. In [8], G2V was deemed secure with respect to phishing because they only considered passive phishing, where the attacker cannot for instance forward the query of the verification code to the user. We believe that it is necessary to consider active phishing as it is a reasonable capability nowadays. This kind of attack can be performed on a large scale without targeting a specific user. We argue that second factor authentication should efficiently protect against phishing, and even phishing combined with fingerprint spoofing, which are likely scenarios under which a user may wish to perform a secure login. Ideally, a second factor should even provide protection against this attack for a completely untrained human only following basic instructions.

Attack Validation. The core of this phishing attack is a *man-in-the-middle* attack. Interestingly, the interception can be completely invisible for the user. In our different examples, we will consider a user who wishes to login on *google.com*. Several user behaviors may be problematic when dealing with phishing:

- the user follows any untrusted link close enough to the authentic one, e.g *google-security.com*;
- the user ignores an HTTPS warning;
- the user does not check that the protocol is HTTPS, but accepts HTTP.

We believe that most untrained users may be victims of the first two, and that even trained users do not always check that they are protected by HTTPS before providing their credentials. Depending on the attacker capabilities, many different kinds of phishing attacks might be performed, some of them difficult to avoid even for experienced users.

The most basic phishing attack is to get the user to click on a malicious link which is close to the official one. It can be performed for instance through a mail which invites the user to login on *google-security.com* to solve some security issue. Here, the attacker may have obtained a valid certificate for the malicious domain, and the user will see a valid HTTPS connection.

Suppose we connect to a malicious WiFi Network. Different kinds of attacks can be performed. First, the malicious network may act as a free WiFi Hotspot network which requires third party authentication. Changing the DNS, for instance to contain the line "google.com IN A 192.168.0.1", the *google.com* domain will be redirected to an IP address controlled by the attacker. This may or may not raise an HTTPS error depending on the state of the cache of the user's browser. More precisely, as most websites, *http://google.com* contains a 301 redirect code to *https://google.com*. This redirection is cached by the browser according to the headers, which contain "Cache-Control: max-age=2592000". This means that the 301 redirect from HTTP to HTTPS is cached by the user browser for 2592000 seconds, i.e., 30 days. If the cache is still valid when the user connects to *google.com*, their browser remembers the 301 and connects to *https://google.com*. As the attacker cannot provide a valid TLS certificate the user sees an HTTPS warning, that the user may choose to ignore. If the cache has expired, which happens once every month, the user connects through HTTP to the malicious server, and believes to be on *google.com* without any warning displayed. If the user does not check for HTTPS, the phishing attempt succeeds.

To confirm this behavior, we forced the DNS client of a Linux machine to resolve the URL "google.com" to a local IP address (editing the */etc/hosts* files). Then, when trying to connect to "http://google.com" in a completely fresh browser session, a local dummy page was displayed without any warning. In a browser session that was used previously to visit the honest Google website in the past 30 days, we obtained the HTTPS warning as the browser remembered the 301 redirect code.

We can design an even stronger attack, by setting up the WiFi network as a network which requires authentication through a captive portal. This is a feature classically supported by most access points, which can be provided with an URL or an IP address to which all users who try to login should be redirected. When performing an actual test for instance on Firefox, the browser detects that we are on a network which requires authentication, and proposes in a pop-up to redirect us to the captive portal. Even a trained user is likely to follow the link to have an Internet connection. The attacker can then redirect the user to a link of their choice: the attacker may redirect the user to *https://google-security.com* with a valid HTTPS certificate, or redirect through basic HTTP to a subdomain of *google.com* that does not exist, for instance *api.login.google.com*, and reconfigure the DNS as previously. As the subdomain does not exist, the attacker is ensured that the user's browser does not have any cached 301 redirection for this site. The user then connects to the attacker server via HTTP on a seemingly legitimate URL. Using a DNS redirection such that all websites are resolved to the captive portal (this is a classical implementation of captive portals for WiFi hotspots), we were able to redirect the user to an arbitrary page, containing any arbitrary link, notably to a fake google page. The fake page corresponding to *http://api.login.google.com* was successfully displayed without warning. To complete the attack with *https://google-security.com*, we would need to register a TLS certificate for this domain. This could have be done for instance using the "Let's encrypt" certification system, although we did not perform this registration.

Some attacks could be avoided or at least complicated through the use of DNSSEC (which enforces a signature validation system for DNS requests) or HSTS (which declares that some website should only be accessed through HTTPS), but this is not supported by most websites, including *google.com*.

The phishing attacks can be made perfect (the user sees *google.com* under HTTPS but is connected to the attacker server) if the attacker can install a malicious HTTPS certificate on the user computer. On a computer running a Debian Linux distribution with libnss3-tools installed, this was achieved through the command *certutil -d sql:$ HOME/.pki/nssdb -A -t TC -n "mitm" -i malicious_cert.pem*. We then successfully reproduced a valid HTTPS connection over a malicious server by using the *mitmproxy* tool on Linux that allows to intercept all connections and mimic the behavior of a man-in-the-middle.

6.4 Action confusion and mixing on *Google 2-step* and U2F

Outline. The different steps of the attack are as follows:

(1) the user initiates an untrusted login;
(2) the attacker transforms the untrusted login into a trusted one;
(3) the attacker uses the acquired cookie to perform other logins;

or

(1) the user initiates an untrusted login;
(2) the attacker initiates a trusted login;
(3) the attacker uses the multi-factor actions made for the untrusted login to validate their session.

Comments. Using multi-factor authentication, a user may expect that after a successful login, once the user has disconnected from the computer, even an attacker with full control over the malicious computer should not be able to perform other logins. If the attacker can obtain a cookie through transforming an untrusted login into a trusted login, this property is violated. Note that this change may be completely invisible to the user, and hence the user may not check the list of trusted devices in the preferences of their account.

The core of this attack is an action confusion, where an intended action, the untrusted login, is transformed into another action, a trusted login. Another instance of action confusion occurs when a verification code intended for a login attempt is used by the attacker to reset the user's password. Note that every SMS from Google has the same content, independent of the action type. We recommend the SMS or the display of the second factor to provide the user with the intended action that is currently being validated. Untrusted login, Trusted login, Password Reset and deactivation of Multi-factor authentication are sensitive actions that should require multi-factor authentication and not be confusable.

This is particularly complicated for U2F, which does not provide the user with feedback through the second factor.

Attacker Validation. We were able to perform several sequences of actions that can lead to action confusion.

First, after a successful second factor login, multi factor parameters of the account can be accessed by only retyping the password. Hence, the second factor protection can be disabled. Although an e-mail is sent to notify the user about this change, as the attacker is already logged in, the attacker can simply delete the e-mail. Once the second factor protection is disabled, the attacker can login from any untrusted computer. This is not *per se* an action confusion, but we note that a code intended for a login also allows an attacker to change the authentication settings of the user.

Second, many browser extensions, e.g., *Greasemonkey* or *TamperMonkey*, allow the behavior of specific pages to be changed. On the Google login page, we were able thanks to a five line JavaScript code to hide the *"I trust this computer"* check-box by adding the attribute *style="visibility:hidden;"* to its *"div"*. As the box is checked by default, we performed a login with a second factor enabled where the box was invisible. The platform then became trusted, and we were able to perform a second login where only the password was required, and the second factor was not asked.

Third, we recall that a login validation and a password reset action yield the same SMS. By initiating simultaneously a login attempt and a password reset, we receive two similar SMS. An attacker may thus initiate a password reset while the user is trying to log into their account. The user will receive the code for the password reset, that the attacker may intercept, e.g., using a key-logger, and change the user's password. We reproduced those attacks using a dedicated Google account and a phone as a second factor.

6.5 USB attack on U2F

Outline. The different steps of the attack are as follows:

(1) the user initiates a login;
(2) the attacker initiates another login;
(3) the attacker sends to the token the payload corresponding to their session;
(4) the attacker sends to the token the payload corresponding to the user session;
(5) the user presses once the button of the token;
(6) the attacker get backs the signed data, and completes their login;
(7) after the first press, the token keeps blinking without any change;
(8) the user presses again, and validates their session.

Comments. This weakness is inherent to the fact that the U2F factor does not provide feedback to the user. Therefore, the user is unable to know which action is actually validated when pressing the token button. Moreover, when submitted two queries in a row, the token will simply keep blinking after the first press. Given that at least some tokens have touch buttons (and not a press button) the user may have the impression that the press was unsuccessful.

The weakness is also related to the fact that U2F does not have an independent communication channel with the server, and cannot provide any security when plugged into a malicious computer. We believe however that providing a secure one time only login on a malicious computer is a reasonable user expectation.

Attack Validation. We performed our tests on the Yubikey U2F FIDO security key, which is equipped with a touch button. Using an open source API[3], we were able to submit two simultaneous signature requests to the token, in a similar way that the requests are submitted by the browser. The token then started blinking as usual, expecting a user touch to confirm the signature. After touching once the button, it kept blinking as if the press were unsuccessful. Pressing once again, we received both signatures through the API. Hence, we could implement a malware that would detect an honest request and would submit a second request at the same time. Then, the user may press once, believe that the press was not registered and thus press once again, thus validating without their consent for the two requests. This problem could be avoided by forbidding two simultaneous requests, and simply dropping any request as long as the current one is not validated.

[3]https://github.com/Yubico/libu2f-server

7 *GOOGLE 2-STEP* VS U2F

7.1 Practical Considerations

As mentioned previously, there are some interesting aspects that are outside of the scope of our threat models and formal analysis. We therefore discuss below some additional thoughts and findings.

Independence of the second factor. When trying to log into an account from a compromised computer, we observed that the U2F token might be used by the attacker if the attacker controls the channel used for communication with the second factor. Therefore, the U2F approach cannot provide strong protection against malwares on the user computer. The risk is mitigated by the fact that the attacker may only perform a single action authenticated by the second factor, but if this action can be used to deactivate the second factor, or reset the user password, the user account may be completely compromised by this single action. The approach of *Google 2-step* provides a second communication channel that is independent from the computer, and may enable security even on a completely untrusted computer.

On the need for feedback. An advantage of the phone over the U2F token is the feedback provided to the user. In particular, on *FIDO's U2F*, two consecutive button presses may remain unnoticed. On the phone, a success or failure confirmation after pressing the button is easily provided. Moreover, the phone can be used to produce improved versions of U2F, where the display is enriched with additional information, as we did for G2DT^{dis}. We note that FIDO proposes the "secure transaction" mechanism, which specifies that second factors might use a display. However, the message content is not included in the standardization.

Storing the keys on a dedicated secure token. An advantage of the U2F token is that, even if a computer is compromised, the number of attacker logins is limited by the number of times the button is pressed. This is due to the fact that keys are stored on a token and not completely compromised. If keys are stored on a computer or a smartphone, a malware may extract them. As discussed previously, U2F does not provide perfect security either. Although keys are more difficult to compromise, one should be careful about how the token is used to ensure that no unwanted computer becomes trusted, or that a user does not press the button twice in a row. A solution to mitigate key leakage for computers or smartphones could be to consider an Isolated Execution Environment, such as Intel SGX, ARM TrustZone or a Trusted Platform Module.

Carrying additional authenticators. An important aspect of multi-factor protocols is of course usability. From that point of view, the need to buy and carry an additional token may be cumbersome. Nowadays, more and more people possess and constantly carry their phone, making it a natural choice for a second factor.

Disabling the second factor. On some websites, for instance GitHub, disabling the second factor (and then changing the password) does not require the use of the second factor, once a login was performed. It seems advisable to require a second factor authentication to disable the mechanism.

7.2 Final comparison

It is difficult to compare the two approaches which are quite different. We try to provide a brief summary of the main advantages of both (we consider here U2F with a dedicated USB token):

- *Google 2-step* provides an independent channel of communication with the server, and feedback through the display;

- *Google 2-step* may be compromised by malware on the phone;
- U2F may provide privacy, if used correctly;
- U2F may suffer from key leakage or device cloning;
- U2F requires an additional device;
- U2F does not provide enough feedback.

If we consider U2F where the phone is used as the dedicated token to store the keys and perform the cryptographic operations, U2F may provide enough feedback to the user (fingerprint, trusted login attempt,...) and would not require carrying another device. We would however potentially lose the privacy, the key storage would need to be completely secured and isolated, it could be a victim of malware, and we need a convenient mechanism to set up a channel between a phone and a computer.

Against both versions of U2F, *Google 2-step* provides better security against some critical scenarios (connection to a dishonest network or on a corrupted computer). Yet, *Google 2-step* is currently unable to provide unlinkability.

8 CONCLUSION

In this paper we propose a detailed threat model for multi-factor authentication protocols. It takes into account communication through TLS channels in an abstract way, yet modelling interesting details such as session identifiers and TLS sessions with compromised agents. Moreover, we consider different levels of malwares in a systematic way by representing a system as a set of interfaces with access rights. Additionally, we allow the adversary to perform phishing and spoof fingerprints, and consider scenarios where a careless user does not perform expected checks. We formalize this model in the applied pi calculus and use the Proverif tool to systematically and automatically analyse several versions of *Google 2-step* and U2F in an extensive way, considering all combinations of threats. The resulting protocol comparison highlights strengths and weaknesses of the different mechanisms, and allows us to propose some simple variants, adding actions to the displayed information or linking the URL to the payload, which improves security. Finally, we validate our models and findings by demonstrating the feasibility of several attacks, in laboratory conditions.

As a direction for future work, it would be interesting to perform an in depth analysis of U2F [11] and the FIDO2 [6], also known as WebAuthn, standards, using our fully mechanized approach.

As another direction, we consider the use of *enclaves* in trusted execution environments: such environments could provide execution certification and a way to enable secure login on a completely untrusted computer, if the computer is equipped with a trusted module. One could then use a phone as a U2F token assuming that we also have an efficient way to establish a channel between the computer and the phone in order to pass the payload. The U2F keys could be stored on the phone, and the next natural step would be to merge G2DT^{dis} and U2F by performing a U2F on the phone in parallel of the G2DT^{dis}. The user would only see the G2DT^{dis} part, which would even be simplified without the double tap, because thanks to the channel between the phone and the computer, there would not be any need to ask the user to select the correct random. G2DT^{dis} combined with for instance the storage of the keys using a trusted execution environment, such as TrustZone would then palliate the issue of keys being revealed due to malware on the phone.

REFERENCES

[1] Martín Abadi, Bruno Blanchet, and Cédric Fournet. 2017. The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication. *J. ACM* 65, 1, Article 1 (Oct. 2017), 41 pages.

[2] Martín Abadi and Cédric Fournet. 2001. Mobile Values, New Names, and Secure Communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*. ACM, New York, NY, USA, 104–115. https://doi.org/10.1145/360204.360213

[3] Alessandro Armando, Roberto Carbone, and Luca Zanetti. 2013. Formal Modeling and Automatic Security Analysis of Two-Factor and Two-Channel Authentication Protocols. In *Network and System Security: 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings*, Javier Lopez, Xinyi Huang, and Ravi Sandhu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 728–734. https://doi.org/10.1007/978-3-642-38631-2_63

[4] D. Basin, S. Radomirovic, and L. Schmid. 2016. Modeling Human Errors in Security Protocols. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. 325–340. https://doi.org/10.1109/CSF.2016.30

[5] David A. Basin, Sasa Radomirovic, and Michael Schläpfer. 2015. A Complete Characterization of Secure Human-Server Communication. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*. 199–213. https://doi.org/10.1109/CSF.2015.21

[6] Vijay Bharadwaj, Hubert Le Van Gong, Dirk Balfanz, Alexei Czeskis, Arnar Birgisson, Jeff Hodges, Michael B. Jones, Rolf Lindemann, and J.C. Jones. 2017. Web Authentication: An API for accessing Public Key Credentials. https://www.w3.org/TR/2017/WD-webauthn-20171205/

[7] Bruno Blanchet. 2016. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Foundations and Trends® in Privacy and Security* 1, 1-2 (2016), 1–135. https://doi.org/10.1561/3300000004

[8] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. 2012. The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, 553–567. https://doi.org/10.1109/SP.2012.44

[9] D. Dolev and A.C. Yao. 1981. On the Security of Public Key Protocols. In *Proc. 22nd Symp. on Foundations of Computer Science (FOCS'81)*. IEEE Comp. Soc. Press, 350–357.

[10] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2014. An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System. In *35th IEEE Symposium on Security and Privacy (S&P 2014)*. IEEE Computer Society, 673–688. https://publ.sec.uni-stuttgart.de/fettkuestersschmitz-sp-2014.pdf

[11] FIDO. 2018. Universal 2nd Factor (U2F). https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/FIDO-U2F-COMPLETE-v1.2-ps-20170411.pdf

[12] Google 2018. Google 2 Step Verification. Web site. https://www.google.com/landing/2step/ Accessed in January 2018.

[13] Paul A. Grassi, James L. Fenton, Elaine M. Newton, Ray A. Perlner, Andrew R. Regenscheid, William E. Burr, Justin P. Richer, Naomi B. Lefkovitz, Jamie M. Danker, Kristen K. Choong, Yee-Yin Greene, and Mary F. Theofanos. 2017. NIST Special Publication 800-63B – Digital Identity Guidelines – Authentication and Lifecycle Management. Available at https://doi.org/10.6028/NIST.SP.800-63b.

[14] Paul A. Grassi, Michael E. Garcia, and James L. Fenton. 2017. NIST Special Publication 800-63-3 – Digital Identity Guidelines. Available at https://doi.org/10.6028/NIST.SP.800-63-3.

[15] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *meltdownattack.com* (2018). https://spectreattack.com/spectre.pdf

[16] Steve Kremer and Robert Künnemann. 2016. Automated analysis of security protocols with global state. *Journal of Computer Security* 24, 5 (2016), 583–616. https://doi.org/10.3233/JCS-160556

[17] Robert Künnemann and Graham Steel. 2013. *YubiSecure? Formal Security Analysis Results for the Yubikey and YubiHSM*. Springer Berlin Heidelberg, Berlin, Heidelberg, 257–272. https://doi.org/10.1007/978-3-642-38004-4_17

[18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *meltdownattack.com* (2018). https://meltdownattack.com/meltdown.pdf

[19] Robert Morris and Ken Thompson. 1979. Password security: A case history. *Commun. ACM* 22, 11 (1979), 594–597.

[20] Olivier Pereira, Florentin Rochet, and Cyrille Wiedling. 2017. Formal Analysis of the Fido 1.x Protocol. In *The 10th International Symposium on Foundations & Practice of Security (Lecture Notes in Computer Science (LNCS))*. Springer.

[21] Andrey Popov, Magnus Nystrom, Dirk Balfanz, Adam Langley, Nick Harper, and Jeff Hodges. 2018. *Token Binding over HTTP*. Internet-Draft draft-ietf-tokbind-https-12. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-tokbind-https-12 Work in Progress.

[22] Source files 2018. Proverif source files and scripts. https://gitlab.inria.fr/cjacomme/multi-factor-authentication-proverif-examples

[23] G Suite team. 2017. G Suite updates. Blog entry. https://gsuiteupdates.googleblog.com/2017/02/improved-phone-prompts-for-2-step.html

[24] Yubico 2018. FIDO Yubikey. Web site. https://www.yubico.com/solutions/fido-u2f/ Accessed in January 2018.

A GLOBAL RESULTS

We summarize in Table 9, 10, an 11 all the results we computed using the automated generation of scenarios, the captions being given in Figure 14. The results were obtained in 8 minutes of computing on a server with 12 Intel(R) Xeon(R) CPU X5650 @ 2.67GHz and 50Gb of RAM. During the computation, 6172 calls to Proverif were made. As Proverif may not terminate we set a timeout at 3 seconds: only two scenarios exceeded the timeout limit. For readability, we only display the minimal or maximal interesting scenarios, and results which are implied by an other scenario are greyed. The table was completely generated by an automated script, to avoid transcription mistakes.

Protocols

- G2V- *Google 2-step* with Verification code
- G2V^{fpr}- G2V with fingerprint display
- G2V^{dis}- G2V^{fpr} with action display
- G2OT- *Google 2-step* One Tap
- G2OT^{fpr}- G2OT with fingerprint display
- G2OT^{dis}- G2OT with action display
- G2DT^{fpr}- *Google 2-step* Double Tap (random to compare)
- G2DT^{dis}- G2DT^{fpr} with action display
- U2F- *FIDO's U2F*
- U2F_{tb}- TokenBinding U2F
- $\text{G2DT}^{\text{dis}}_{\text{tb}}$- TokenBinding G2DT^{dis}

Scenarios:

- NC- No Compare, the human does not compare values
- FS- Fingerprint spoof, the attacker can copy the user IP address
- PH- The user might be victim of phishing only on trusted everyday connections or untrusted computers
- $\mathcal{M}^{\text{interface}}_{\text{in:}acc1,\text{out:}acc2}$- The interface inputs are given to the attacker with access level acc1, and acc2 for the outputs

Notations:

- ✓- Property satisfied (✔if all three)
- ✗- Attack found (✖if all three)
- ✗- Attack also present in a weaker scenario
- ✓- Property also satisfied in a stronger scenario
- - - Either scenario not pertinent, or failure to reconstruct attack trace

Fig. 14. Caption for global results

Threat Scenarios				g2V	g2VFPR	g2VD	g2ST	g2STFPR	g2STD	g2DTFPR	g2DTD	g2DTDE	U2F	TB-U2F	TB-g2DTD
				✔	✔	✔	✖	✔	✔	✔	✔	✔	✔	✔	✔
PH				✖	✔	✓✓✓	✖	✔	✔	✔	✓✓✓	✔	✔	✔	✓✓✓
		NC		✔	✔	✔	✖	✖	✖	✔	✔	✔	✔	✔	✔
	FS			✔	✔	✔	✖	✖	✖	✔	✔	✔	✔	✔	✔
PH		NC		✖	✖	✖	✖	✖	✖	✖	✖	✓✓✓	✔	✔	✖
PH		NC	$\mathcal{M}^{\text{t-dis}}_{\text{io:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✓	✓✓✓	✖
PH	FS			✖	✖	✗✓✓	✖	✖	✖	✖	✗✓✓	✔	✔	✔	✗✓✓
PH	FS		$\mathcal{M}^{\text{t-mem}}_{\text{in:}\mathcal{RO}}$	✖	✖	✗✓✗	✖	✖	✖	✖	✗✓✗	✓✓-	✓✓✗	✓✓✗	✗✓✗
PH	FS		$\mathcal{M}^{\text{t-tls}}_{\text{io:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✗✓-	✓✓✗	✓✓✗	✔	✗✓✓
PH	FS		$\mathcal{M}^{\text{t-usb}}_{\text{in:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✗✓✓	✔	✔	✔	✗✓✓
PH	FS	NC	$\mathcal{M}^{\text{t-mem}}_{\text{in:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓-	✓✓✗	✓✓✗	✖
PH	FS	NC	$\mathcal{M}^{\text{t-tls}}_{\text{io:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✓✓✗	✓✓✓	✖
PH	FS		$\mathcal{M}^{\text{t-mem}}_{\text{in:}\mathcal{RW}}$	✖	✖	✗✓✗	✖	✖	✖	✖	✗✓✗	✓✓✗	✓✓✗	✓✓✗	✗✓✗
PH	FS		$\mathcal{M}^{\text{t-tls}}_{\text{io:}\mathcal{RO}}\ \mathcal{M}^{\text{t-mem}}_{\text{in:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✗✓-	✓✓✗	✓✓✗	✓✓✗	✗✓✗
PH	FS		$\mathcal{M}^{\text{t-dis}}_{\text{io:}\mathcal{RW}}$	✖	✖	✗✓✓	✖	✖	✖	✖	✖	✓✗✗	✔	✔	✖
PH	FS	NC	$\mathcal{M}^{\text{t-mem}}_{\text{in:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✓✓✗	✓✓✗	✖
PH	FS		$\mathcal{M}^{\text{t-usb}}_{\text{in:}\mathcal{RO}}\ \mathcal{M}^{\text{t-mem}}_{\text{in:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✗✓✗	✓✓-	✓✓✗	✓✓✗	✗✓✗
PH	FS		$\mathcal{M}^{\text{t-tls}}_{\text{io:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✖	✖	✗✗-
PH	FS	NC	$\mathcal{M}^{\text{t-tls}}_{\text{io:}\mathcal{RO}}\ \mathcal{M}^{\text{t-mem}}_{\text{in:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✓✓✗	✓✓✗	✖
PH	FS		$\mathcal{M}^{\text{t-usb}}_{\text{in:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✔	✓✓✓	✔	✖
PH	FS		$\mathcal{M}^{\text{t-dis}}_{\text{io:}\mathcal{RW}}\ \mathcal{M}^{\text{t-mem}}_{\text{in:}\mathcal{RO}}$	✖	✖	✗✓✗	✖	✖	✖	✖	✖	✓✗✗	✓✓✗	✓✓✗	✖
PH	FS		$\mathcal{M}^{\text{t-usb}}_{\text{in:}\mathcal{RO}}\ \mathcal{M}^{\text{t-mem}}_{\text{in:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✗✓✗	✓✓✗	✓✓✗	✓✓✗	✗✓✗
PH	FS		$\mathcal{M}^{\text{t-dis}}_{\text{io:}\mathcal{RW}}\ \mathcal{M}^{\text{t-tls}}_{\text{io:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✗✗	✓✓✗	✓✓✓	✖
PH	FS		$\mathcal{M}^{\text{t-tls}}_{\text{io:}\mathcal{RW}}\ \mathcal{M}^{\text{t-mem}}_{\text{in:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✖	✖	✖
PH	FS	NC	$\mathcal{M}^{\text{t-dis}}_{\text{io:}\mathcal{RW}}\ \mathcal{M}^{\text{t-mem}}_{\text{in:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✓✓✗	✖
PH	FS		$\mathcal{M}^{\text{t-usb}}_{\text{in:}\mathcal{RO}}\ \mathcal{M}^{\text{t-dis}}_{\text{io:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✗✗	✓✓✓	✓✓✓	✖
PH	FS		$\mathcal{M}^{\text{t-dis}}_{\text{io:}\mathcal{RO}}\ \mathcal{M}^{\text{t-tls}}_{\text{io:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✖	✖	✖
PH	FS		$\mathcal{M}^{\text{t-usb}}_{\text{in:}\mathcal{RW}}\ \mathcal{M}^{\text{t-mem}}_{\text{in:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✖	✖	✖
PH	FS		$\mathcal{M}^{\text{t-usb}}_{\text{in:}\mathcal{RW}}\ \mathcal{M}^{\text{t-tls}}_{\text{io:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✖	✖	✖
PH	FS		$\mathcal{M}^{\text{t-usb}}_{\text{io:}\mathcal{RW}}\ \mathcal{M}^{\text{t-dis}}_{\text{io:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✗✗	✖	✖	✖
			$\mathcal{M}^{\text{t-mem}}_{\text{in:}\mathcal{RO}}$	✓✓✓	✓✓✓	✓✓✓	✖	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓
			$\mathcal{M}^{\text{dev}}_{\text{in:}\mathcal{RO}}$	✖	✖	✖	✖	✔	✔	✔	✔	✔	---	---	✔
			$\mathcal{M}^{\text{t-dis}}_{\text{io:}\mathcal{RO}}$	✔	✔	✔	✖	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✔	✔	✓✓✓
			$\mathcal{M}^{\text{t-tls}}_{\text{io:}\mathcal{RO}}$	✖	✓✓✓	✓✓✓	✖	✓✓✓	✓✓✓	✔	✔	✔	✔	✔	✔
			$\mathcal{M}^{\text{t-usb}}_{\text{in:}\mathcal{RO}}$	✖	✔	✔	✖	✔	✔	✔	✔	✔	✔	✔	✔
		NC	$\mathcal{M}^{\text{t-tls}}_{\text{io:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✓✓✓	✓✓✓	✓✓✓	✔	✔	✔
		NC	$\mathcal{M}^{\text{t-usb}}_{\text{in:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✓✓✓	✓✓✓	✔	✔	✔	✓✓✓
			$\mathcal{M}^{\text{dev}}_{\text{in:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
			$\mathcal{M}^{\text{dev}}_{\text{io:}\mathcal{RO}}$	✖	✖	✖	✖	✔	✔	✔	✔	✔	✖	✖	✔
			$\mathcal{M}^{\text{t-tls}}_{\text{io:}\mathcal{RW}}$	✖	✗✓✗	✓✓✗	✖	✗✓✗	✓✓✗	✗✓✗	✓✓✗	✓✓✗	✖	✗✗-	✓✓✗
		NC	$\mathcal{M}^{\text{t-dis}}_{\text{io:}\mathcal{RW}}$	✓✓✓	✓✓✓	✓✓✓	✖	✖	✖	✖	✖	✖	✔	✔	✖
		NC	$\mathcal{M}^{\text{t-tls}}_{\text{io:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✗✗✗	✗✗✗	✓✓✗	✖	✗✗-	✗✗✗
		NC	$\mathcal{M}^{\text{t-usb}}_{\text{in:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✔	✓✓✓	✔	✖
			$\mathcal{M}^{\text{t-tls}}_{\text{io:}\mathcal{RW}}\ \mathcal{M}^{\text{t-mem}}_{\text{in:}\mathcal{RO}}$	✖	✗✓✗	✓✓✗	✖	✗✓✗	✓✓✗	✗✓✗	✓✓✗	✓✓✗	✖	✗✗✗	✓✓✗
			$\mathcal{M}^{\text{t-dis}}_{\text{io:}\mathcal{RO}}\ \mathcal{M}^{\text{t-tls}}_{\text{io:}\mathcal{RW}}$	✖	✗✓✗	✓✓✗	✖	✗✓✗	✓✓✗	✗✓✗	✓✓✗	✓✓✗	✖	✗✗✗	✓✓✗
		NC	$\mathcal{M}^{\text{t-tls}}_{\text{io:}\mathcal{RW}}\ \mathcal{M}^{\text{t-mem}}_{\text{in:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✖	✖	✖

Table 9. Global results for malware on trusted platform part 1

		Threat Scenarios	g2V	g2VFPR	g2VD	g2ST	g2STFPR	g2STD	g2DTFPR	g2DTD	g2DTDE	U2F	TB-U2F	TB-g2DTD
		$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}\ \mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✖	✗✓✗	✓✓✗	✖	✗✓✗	✓✓✗	✗✓✗	✓✓✗	✓✓✗	✖	✗✗✗	✓✓✗
	NC	$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}\ \mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✓	✔	✖
	NC	$\mathcal{M}^{t-dis}_{io:\mathcal{RO}}\ \mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✖	✖	✖
	NC	$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}\ \mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✓	✓✓✓	✖
	NC	$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}\ \mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✖	✖	✖
		$\mathcal{M}^{t-usb}_{io:\mathcal{RW}}$	✖	✔	✓✓✓	✖	✔	✔	✔	✓✓✓	✔	✖	✖	✓✓✓
	NC	$\mathcal{M}^{t-usb}_{io:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✓	✖	✖	✖
FS		$\mathcal{M}^{t-mem}_{in:\mathcal{RO}}$	✓✓✗	✓✓✗	✓✓✗	✖	✖	✖	✓✓✗	✓✓✗	✓✓-	✓✓✗	✓✓✗	✓✓✗
FS		$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✓✗✗	✖	✖	✖	✓✓-	✓✓-	✓✓✗	✓✓✗	✔	✔
FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$	✖	✖	✓✗✗	✖	✖	✖	✓✓✓	✔	✔	✔	✔	✔
FS	NC	$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✓✓-	✓✓-	✓✓✗	✓✓✗	✔	✓✓✓
FS		$\mathcal{M}^{t-mem}_{in:\mathcal{RW}}$	✓✓✗	✓✓✗	✓✓✗	✖	✖	✖	✓✓✗	✓✓✗	✓✓✗	✓✓✗	✓✓✗	✓✓✗
FS		$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}\ \mathcal{M}^{t-mem}_{in:\mathcal{RO}}$	✖	✖	✓✗✗	✖	✖	✖	✓✓-	✓✓-	✓✓✗	✓✓✗	✓✓✗	✓✓✗
FS		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✓✓✓	✓✓✓	✔	✖	✖	✖	✖	✓✗✗	✓✗✗	✔	✔	✓✗✗
FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}\ \mathcal{M}^{t-mem}_{in:\mathcal{RO}}$	✖	✖	✓✗✗	✖	✖	✖	✓✓✗	✓✓✗	✓✓-	✓✓✗	✓✓✗	✓✓✗
FS		$\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✖	✖	✓✗✗	✖	✖	✖	✗✗✗	✓✗✗	✓✓✗	✖	✗✗✗	✓✗-
FS	NC	$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}\ \mathcal{M}^{t-mem}_{in:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✓✓-	✓✓-	✓✓✗	✓✓✗	✓✓✗	✓✓✗
FS	NC	$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}\ \mathcal{M}^{t-mem}_{in:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✓✓✗	✓✓✗	✓✓-	✓✓✗	✓✓✗	✓✓✗
FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗✗	✔	✔	✔	✓✗✗
FS	NC	$\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✖	✖	✖
FS		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}\ \mathcal{M}^{t-mem}_{in:\mathcal{RO}}$	✓✓✗	✓✓✗	✓✓✗	✖	✖	✖	✖	✓✗✗	✓✗✗	✓✓✗	✓✓✗	✓✗✗
FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}\ \mathcal{M}^{t-mem}_{in:\mathcal{RW}}$	✖	✖	✓✗✗	✖	✖	✖	✓✓✗	✓✓✗	✓✓✗	✓✓✗	✓✓✗	✓✓✗
FS		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}\ \mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗✗	✓✗✗	✓✓✗	✔	✓✗✗
FS		$\mathcal{M}^{t-tls}_{io:\mathcal{RW}}\ \mathcal{M}^{t-mem}_{in:\mathcal{RO}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗✗	✓✓✗	✓✓✗	✓✓✗	✓✗✗
FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}\ \mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗✗	✓✗✗	✔	✔	✓✗✗
FS	NC	$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}\ \mathcal{M}^{t-mem}_{in:\mathcal{RO}}$	✓✓✗	✓✓✗	✓✓✗	✖	✖	✖	✖	✖	✖	✓✓✗	✓✓✗	✖
FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}\ \mathcal{M}^{t-mem}_{in:\mathcal{RO}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗✗	✓✓✗	✓✓✗	✓✓✗	✓✗✗
FS	NC	$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}\ \mathcal{M}^{t-mem}_{in:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✓✓✗	✓✓✗	✓✓✗	✓✓✗	✓✓✗	✓✓✗
FS	NC	$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}\ \mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✓✓✓	✖
FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}\ \mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗✗	✓✓✗	✓✓✗	✔	✓✗✗
FS	NC	$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}\ \mathcal{M}^{t-mem}_{in:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✓✓✗	✓✓✗	✓✓✗
FS		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}\ \mathcal{M}^{t-tls}_{io:\mathcal{RO}}\ \mathcal{M}^{t-mem}_{in:\mathcal{RO}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗✗	✓✗✗	✓✓✗	✓✓✗	✓✗✗
FS	NC	$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}\ \mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✓✓✗	✔	✖
FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}\ \mathcal{M}^{t-dis}_{io:\mathcal{RW}}\ \mathcal{M}^{t-mem}_{in:\mathcal{RO}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗✗	✓✗✗	✓✓✗	✓✓✗	✓✗✗
FS		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}\ \mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗✗	✓✗✗	✖	✖	✓✗✗
FS	NC	$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}\ \mathcal{M}^{t-tls}_{io:\mathcal{RO}}\ \mathcal{M}^{t-mem}_{in:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✓✓✗	✖
FS	NC	$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}\ \mathcal{M}^{t-dis}_{io:\mathcal{RW}}\ \mathcal{M}^{t-mem}_{in:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✓✓✗	✖
FS		$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}\ \mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗✗	✓✓✗	✖	✖	✓✗✗
FS		$\mathcal{M}^{t-usb}_{io:\mathcal{RW}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗✗	✔	✖	✖	✓✗✗
FS		$\mathcal{M}^{t-usb}_{io:\mathcal{RW}}\ \mathcal{M}^{t-mem}_{in:\mathcal{RO}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗✗	✓✓✗	✖	✖	✓✗✗
FS		$\mathcal{M}^{t-usb}_{io:\mathcal{RW}}\ \mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗✗	✓✓✗	✖	✖	✓✗✗
FS	NC	$\mathcal{M}^{t-usb}_{io:\mathcal{RW}}\ \mathcal{M}^{t-mem}_{in:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✖	✖	✖
FS	NC	$\mathcal{M}^{t-usb}_{io:\mathcal{RW}}\ \mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✖	✖	✖
FS		$\mathcal{M}^{t-usb}_{io:\mathcal{RW}}\ \mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗✗	✓✗✗	✖	✖	✓✗✗

Table 10. Global results for malware on trusted platform, part 2

Threat Scenarios			g2V	g2VFPR	g2VD	g2ST	g2STFPR	g2STD	g2DTFPR	g2DTD	g2DTDE	U2F	TB-U2F	TB-g2DTD
PH	NC	$\mathcal{M}^{\text{u-dis}}_{\text{io:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✔	✔	✖
PH	FS	$\mathcal{M}^{\text{u-dis}}_{\text{io:}\mathcal{RW}}$	✖	✖	✗✓✓	✖	✖	✖	✖	✗✓✓	✗✓✓	✔	✔	✗✓✓
		$\mathcal{M}^{\text{u-mem}}_{\text{in:}\mathcal{RO}}$	✔	✔	✔	✖	✔	✔	✔	✔	✔	✔	✔	✔
		$\mathcal{M}^{\text{u-dis}}_{\text{io:}\mathcal{RO}}$	✔	✔	✔	✖	✔	✔	✔	✓✓✓	✓✓✓	✔	✔	✓✓✓
		$\mathcal{M}^{\text{u-tls}}_{\text{io:}\mathcal{RO}}$	✖	✓✓✓	✔	✖	✓✓✓	✔	✔	✔	✔	✔	✔	✔
		$\mathcal{M}^{\text{u-usb}}_{\text{in:}\mathcal{RO}}$	✖	✓✓✓	✔	✖	✓✓✓	✔	✔	✔	✔	✔	✔	✔
	NC	$\mathcal{M}^{\text{u-tls}}_{\text{io:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✔	✔	✔	✔	✔	✔
	NC	$\mathcal{M}^{\text{u-usb}}_{\text{in:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✔	✔	✓✓✓	✔	✔	✔
		$\mathcal{M}^{\text{u-tls}}_{\text{io:}\mathcal{RW}}$	✖	✓✗✗	✓✓✓	✖	✓✗✗	✔	✓✗✗	✓✓✓	✔	✖	✖	✓✓✓
	NC	$\mathcal{M}^{\text{u-dis}}_{\text{io:}\mathcal{RW}}$	✔	✔	✔	✖	✖	✖	✖	✖	✖	✔	✔	✖
		$\mathcal{M}^{\text{u-usb}}_{\text{in:}\mathcal{RW}}$	✖	✓✗✓	✔	✖	✓✗✓	✔	✓✗✓	✔	✔	✓✗✓	✓✗✓	✔
	NC	$\mathcal{M}^{\text{u-tls}}_{\text{io:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✗✗✗	✖	✓✓✓	✖	✖	✖
	NC	$\mathcal{M}^{\text{u-usb}}_{\text{in:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✗✗✗	✖	✓✗✓	✓✗✓	✓✗✓	✖
	NC	$\mathcal{M}^{\text{u-dis}}_{\text{io:}\mathcal{RW}}$ $\mathcal{M}^{\text{u-tls}}_{\text{io:}\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✔	✔	✖
	NC	$\mathcal{M}^{\text{u-usb}}_{\text{in:}\mathcal{RO}}$ $\mathcal{M}^{\text{u-dis}}_{\text{io:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✓	✓✓✓	✖
	NC	$\mathcal{M}^{\text{u-usb}}_{\text{in:}\mathcal{RW}}$ $\mathcal{M}^{\text{u-dis}}_{\text{io:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✓✗✓	✓✗✓	✖
		$\mathcal{M}^{\text{u-usb}}_{\text{io:}\mathcal{RW}}$	✖	✓✗✓	✓✓✓	✖	✓✗✓	✔	✓✗✓	✓✓✓	✔	✗✗✗	✗✗✗	✓✓✓
	NC	$\mathcal{M}^{\text{u-usb}}_{\text{in:}\mathcal{RW}}$ $\mathcal{M}^{\text{u-tls}}_{\text{io:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✗✗	✖	✖	✖
	NC	$\mathcal{M}^{\text{u-usb}}_{\text{io:}\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✗✓	✖	✖	✖
FS		$\mathcal{M}^{\text{u-tls}}_{\text{io:}\mathcal{RO}}$	✖	✖	✗✓✓	✖	✖	✖	✔	✓✓✓	✔	✔	✔	✓✓✓
FS		$\mathcal{M}^{\text{u-usb}}_{\text{in:}\mathcal{RO}}$	✖	✖	✗✓✓	✖	✖	✖	✔	✓✓✓	✔	✔	✔	✓✓✓
FS		$\mathcal{M}^{\text{u-dis}}_{\text{io:}\mathcal{RW}}$	✔	✔	✓✓✓	✖	✖	✖	✖	✗✓✓	✗✓✓	✔	✔	✗✓✓
FS		$\mathcal{M}^{\text{u-tls}}_{\text{io:}\mathcal{RW}}$	✖	✖	✗✓✓	✖	✖	✖	✗✗✗	✗✓✓	✓✓✓	✖	✖	✗✓✓
FS		$\mathcal{M}^{\text{u-usb}}_{\text{in:}\mathcal{RW}}$	✖	✖	✗✓✓	✖	✖	✖	✗✗✗	✗✓✓	✔	✓✗✓	✓✗✓	✗✓✓
FS		$\mathcal{M}^{\text{u-dis}}_{\text{io:}\mathcal{RW}}$ $\mathcal{M}^{\text{u-tls}}_{\text{io:}\mathcal{RO}}$	✖	✖	✗✓✓	✖	✖	✖	✖	✗✓✓	✗✓✓	✔	✔	✗✓✓
FS		$\mathcal{M}^{\text{u-usb}}_{\text{in:}\mathcal{RO}}$ $\mathcal{M}^{\text{u-dis}}_{\text{io:}\mathcal{RW}}$	✖	✖	✗✓✓	✖	✖	✖	✖	✗✓✓	✗✓✓	✓✓✓	✓✓✓	✗✓✓
FS		$\mathcal{M}^{\text{u-dis}}_{\text{io:}\mathcal{RW}}$ $\mathcal{M}^{\text{u-tls}}_{\text{io:}\mathcal{RW}}$	✖	✖	✗✓✓	✖	✖	✖	✖	✗✓✓	✗✓✓	✓✗✓	✓✗✓	✗✓✓
FS		$\mathcal{M}^{\text{u-usb}}_{\text{in:}\mathcal{RW}}$ $\mathcal{M}^{\text{u-dis}}_{\text{io:}\mathcal{RW}}$	✖	✖	✗✓✓	✖	✖	✖	✖	✗✓✓	✗✓✓	✓✗✓	✓✗✓	✗✓✓
FS		$\mathcal{M}^{\text{u-usb}}_{\text{io:}\mathcal{RW}}$	✖	✖	✗✓✓	✖	✖	✖	✖	✗✓✓	✓✓✓	✖	✖	✗✓✓
FS		$\mathcal{M}^{\text{u-usb}}_{\text{io:}\mathcal{RW}}$ $\mathcal{M}^{\text{u-dis}}_{\text{io:}\mathcal{RW}}$	✖	✖	✗✓✓	✖	✖	✖	✖	✗✓✓	✗✓✓	✖	✖	✗✓✓

Table 11. Global results for malwares on untrusted platform