An extensive formal analysis of multi-factor authentication protocols

Charlie Jacomme
LSV & CNRS & ENS Paris-Saclay
& Inria & Université Paris-Saclay

Steve Kremer
LORIA, Inria Nancy-Grand Est
& CNRS & Université de Lorraine

***Abstract*—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 used 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*. 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. Our analysis highlights weaknesses and strengths of the different protocols, and allows us to suggest several small modifications of the existing protocols which are easy to implement, yet improve their security in several threat scenarios.**

1. Introduction

Users need to authenticate to an increasing number of electronic services in everyday life: emails, agenda, bank accounts, 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"* [1]. 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 [2]. Since then, things have become worse: many people choose the same weak passwords for many 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 [3] 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 his mobile phone, or a dedicated authentication token. Two popular protocols are *Google 2-step* [4] (which actually regroups several mechanisms) and *FIDO's U2F* [5], 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 his phone that he must copy onto his 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 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 and authentication of messages 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 victim

of *phishing*, or *pharming*, and hence willing to connect to and enter his 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 model these threat scenarios in the applied pi calculus and use the PROVERIF tool to analyse several variants of the *Google 2-step* and *FIDO's U2F* protocols. 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. Even though we eliminate threat scenarios as soon as results are implied by weaker scenarios, the analysis required over 6 000 calls to PROVERIF. Our analysis results in a detailed comparison of the protocols which highlights their respective weaknesses and strengths. It also 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.

Related work. Bonneau et al. [6] 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. 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* [7], the *Yubikey One Time Password* [8], [9] and the *Secure Call Authorization* protocols [10]. However, those analyses do not study resistance to malware, nor do they capture precisely TLS channel behaviour or fingerprints. Basin et al. [11] 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 [11], as they simply check that every successful login was proceeded by an attempt from the corresponding user to login. In the same vein, [12] studies minimal topologies to establish secure channels between humans and servers.

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*.

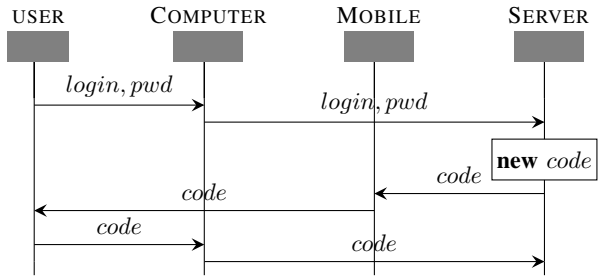

Figure 1: G2V protocol

2.1. *Google 2-step*

To improve security of user logins, Google proposes a two factor authentication mechanism called *Google 2-step* [4]. 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 newer version of "One-Tap" that we dubbed "Double-Tap".

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 user enters his login and password into his computer, which forwards the information to the server. 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. The user then copies the code to his computer, which sends it to the server. If the correct code is received login is granted.

2.1.2. *Google 2-step* with One-Tap - G2OT. In Figure 2 we present the One-Tap version of *Google 2-step*. The protocol starts as the verification code variant with the transmission of the login credentials to the server. 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. The phone displays a pop up to the user who can then confirm the action or abort it, by choosing "Yes" or "No" respectively. 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 [13] that the pop up would also contain in the future a fingerprint of the

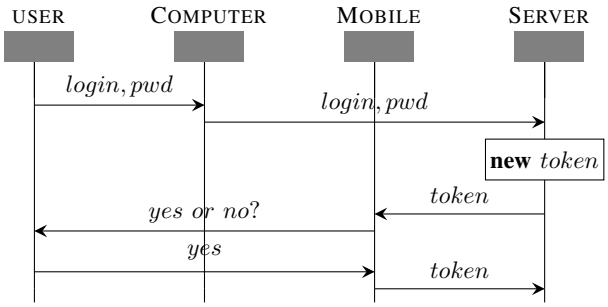

Figure 2: G2OT protocol

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^{fpr}) and without (G2OT) the fingerprint.

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. It is not documented, 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 two other random numbers. The user is then asked to select on his phone the number displayed on his computer. This mimics the behaviour of a verification code displayed on the computer and that the user should enter on his phone, but with the benefits of greater simplicity and ease of use.

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

FIDO is an alliance which aims at providing standards for secure authentication. Among their proposed solutions is the Universal 2nd Factor (U2F) protocol [5]. We here concentrate on the version using a USB token as the second factor. 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 and the public key is registered on the server. This operation is performed once, and the token can then be used for authenticating; the steps of the authentication protocol are presented in Figure 3. First, the computer forwards the user's login and password to the server. Then, the server generates a challenge which is sent to the user's computer. 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. The user confirms the transaction by

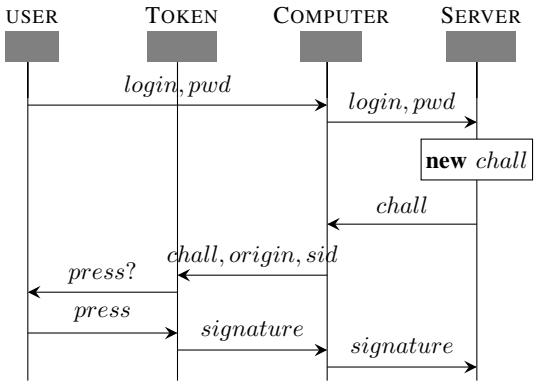

Figure 3: U2F protocol

pressing the token button. The signature is then forwarded to the server for verification.

2.3. Disabling the second factor on trusted devices

When designing an authentication protocol, as also emphasized in [6], 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 his emails on his main laptop would be very cumbersome. This is 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. 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 browsers version, etc. As these 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.

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 [14] 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.

3.1.1. Systems as interfaces. To give a principled model of malwares and what parts of a system a 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.

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 keylogger 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). However, reading data sent by an application to some USB device, 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 5 levels, that can be organized as a lattice depicted in Figure 4.

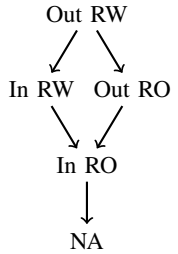

Figure 4: Access

3.1.2. Malware on a computer. For a computer, we will consider three 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 outputs interfaces;
- the TLS interface, capturing the network communications, but by always assuming that the attacker as the same level of control over inputs and outputs.

We can succinctly describe a malware on a computer by giving for each interface inputs and output the attacker access, and we will the use the notation : $\mathcal{M}^{\mathsf{interface}}_{\mathsf{in}:acc1,out:acc2}$, where interface might be tls, usb or dis, and $acc1$ and $acc2$ might be $\mathcal{RO}$ or $\mathcal{RW}$.

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.

3.1.3. Malware on a phone. For a mobile phone, the type of interface may vary from protocols, with for instance SMS inputs or TLS inputs. To simplify, we will only consider a phone to have one input and one output interface. We thus only consider a generic device interface called dev, with all possible access level. $\mathcal{M}^{\mathsf{dev}}_{\mathsf{in}:\mathcal{RO}}$ 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, he provides information about himself, called his *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 because he 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 victim 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 an experienced user may be a victim, 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 him to continue the protocol only if the fingerprint corresponds to his 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 value given to him 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 allowing for a detailed comparison of the different protocols, allowing us 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 connecting to a Wifi hotspot in an hotel or a train station, the Wifi might be controlled by the attacker, and we would have fingerprint spoof and phishing, 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 keylogger ($\mathcal{M}^{\mathsf{usb}}_{\mathsf{in}:\mathcal{RO}}$). However, if we connect on a computer shared by many people at some place, for instance at a cybercaffé, 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 ($\mathcal{M}^{\mathsf{tls}}_{\mathsf{io}:rw}$). 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 a high level of granularity going from no attacker power at all to complete control over both the network and the platforms. Our threat model abstracts 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. For instance, a side channel attack such as Meltdown [15] or Spectre [16] may allow the attacker to read the memory of the user computer: this is captured by giving read-only access to all the interfaces of the computer ($\mathcal{M}^{\mathsf{usb}}_{\mathsf{in}:\mathcal{RO}}$ $\mathcal{M}^{\mathsf{tls}}_{\mathsf{in}:\mathcal{RO}}$ $\mathcal{M}^{\mathsf{dis}}_{\mathsf{in}:\mathcal{RO}}$). Another example 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 [17], [18] that is used as input language by the PROVERIF tool [19] 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 [19] 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 5.

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, bool 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

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

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

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

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

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

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. **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. **new** $n : T$ declares a fresh name of type T; this construct is useful to generate fresh, secret keys and nonces. **in**$(M, x : T)$ inputs a term that will be bound to a variable of type T on channel M and **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 on this channel. The conditional **if** $E_1 = E_2$ **then** P **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.

As an example consider the processes defined in Figure 6. A server process S signs a freshly generated random bitstring rnd and sends it to a user process U. U raises the event $Accept(rnd)$ if the signature is valid. The main process then declares that sk is a fresh secret key and executes an unbounded number of copies of S and U in parallel.

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 for any execution, any 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 6, we model the property that any accepted session with a given random was actually initiated with the same random as

$$Accept(x) \Longrightarrow Init(x)$$

This property is indeed satisfied. However, it may be too weak as it does not capture replay attacks. We may have twice the event $Accept(rnd)$ (for the same value rnd) while this session was only initiated once. To capture such replay attack we may use *injective* correspondence properties

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

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

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. 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 $\mathtt{tls}(\mathtt{id},\mathtt{id}) : \mathtt{channel}$ where `id` is a user defined type of identities, and use the channel $\mathtt{tls}(c,s)$ for communications between client c and server s;
- we define a *TLS manager* process that given as inputs two identities id_1, id_2 and outputs on a public channel the channel name $\mathtt{tls}(id_1,id_2)$, 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.

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 **out**$(\mathtt{tls}(c,s),M).P$ into a process **out**$(\mathtt{tls}(c,s),M)|P$. This ensures that the communications on TLS channels are indeed *asynchronous*.

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

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

Figure 5: Terms and processes

$S(sk : \mathtt{skey}) \hat{=}$
  new $rnd : \mathtt{bitstring}$.
  event $\mathtt{Init}(rnd)$.
  out$(a, (\mathtt{sign}(rnd, sk), rnd))$

$U(pk : \mathtt{pkey}) \hat{=}$
  in$(a, (sig : \mathtt{bitstring}, rnd : \mathtt{bitstring}))$.
  if $\mathtt{checksign}(sig, pk) = rnd$ **then**
    event $\mathtt{Accept}(rnd)$.

new $sk : \mathtt{skey};\ !S(sk) | !U(\mathtt{pk}(sk))$

Figure 6: Process example

modelled by rewriting processes of the form **in**$(ch, x).P$ into processes of the form **in**(ch, x).**out**$(a, x).P$, respectively **out**$(ch, M).P$ into **out**(a, M).**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.

4.3.2. 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 code that he may receive through the display interface of its computer or its 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.

4.3.3. Human errors - Phishing. In our model of TLS we simply represent a url by the server identity idS, provided by the human user. 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 that of a malicious server, which his 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.

4.3.4. Fingerprint and spoofing. As discussed before, when browsing, one may extract informations 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 $\mathtt{fpr}(id) : \mathtt{fingerprint}$ 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 allows the attacker to initiate a communication on a channel $\mathtt{tls}(\mathtt{spoof}_{\mathtt{fpr}}(\mathtt{fpr}(idC)), s)$ using a fingerprint that is identical to $\mathtt{fpr}(idC)$.

5. Analysis and Comparison

In this section we use the formal framework to analyze 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 [20].

5.1. Properties and methodology

5.1.1. Properties. We will concentrate 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 computer that he previously trusted, using his 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

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

The $init_x(\texttt{id})$ events are added to the process modelling the human user, in order to capture the user's intention to perform the login action. The $accept_x$ 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 is matching 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, with for instance $\mathcal{M}^{\mathsf{u-usb}}_{\mathsf{io}:\mathcal{RO}}$ for a keylogger 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 $\mathcal{M}^{\mathsf{t-usb}}_{\mathsf{io}:\mathcal{RW}}$" corresponds to the scenario where the attacker can perform phishing, fingerprint spoofing, and has read-write access to usb devices of the trusted computer, and "NC $\mathcal{M}^{\mathsf{u-tls}}_{\mathsf{io}:\mathcal{RW}}$ $\mathcal{M}^{\mathsf{u-usb}}_{\mathsf{io}:\mathcal{RW}}$ $\mathcal{M}^{\mathsf{u-dis}}_{\mathsf{io}:\mathcal{RW}}$" corresponds to a human that does not perform comparisons and full attacker control (read-write access to TLS, USB and display interfaces) on 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 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 7 and 8 in Appendix.

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

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

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) and give the status of the authentication property for $u$ntrusted login, $t$rusted login and $c$ookie login for the given threat scenario;
- ✖ 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 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 his phone to his 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 keylogger ($\mathcal{M}^{\mathsf{t-usb}}_{\mathsf{in}:\mathcal{RO}}$), or by reading the SMS interface ($\mathcal{M}^{\mathsf{dev}}_{\mathsf{in}:\mathcal{RO}}$), or any other read access on an interface on which the code is transmitted, the attacker can use this code to validate his own session.

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. This may be useful as it allows to print in advance a set of codes, e.g., if no SMS access is available. In theory, linking the code to a session does not improve security, 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*, in which the phone and the server shares a secret key, and use it to derive from the current time a one time password. 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 his 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 Appendix A. While the attack is in our most basic threat mode, it nevertheless requires that the attacker can detect a login attempt from the user, and can block network messages.

5.2.3. G2OT^{fpr}. We provide in the third column of Table 1 the analysis of G2OT^{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 his 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^{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 of FS or NC, G2V provides better guarantees than G2OT^{fpr}. It is however interesting to note that G2OT^{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 also travel over TLS between the relay and the provider's servers. While this argument is in favour of G2OT^{fpr}, one may also argue that G2V has better resistance to user inattention, as a user needs to actively copy a code.

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, where we highlight in blue the differences between G2OT^{fpr} and G2DT^{fpr}. The attacker must now be able to have its code displayed to the user and then selected onto the user's device in order to successfully login, so even in case of FS or NC and some read only access, it is not enough. 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 in practice mount an attack against G2DT^{fpr}.

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

Table 2: Analysis of the *Google 2-step* Double-Taps

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 $\mathcal{M}^{\mathsf{u-usb}}_{\mathsf{in}:\mathcal{RW}}$. If this happens, the attacker can obtain a session cookie for its 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.

Thus, we also add for all the protocols the action the user is trying to perform (trusted or untrusted login) to the display. This mechanism 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 to other types of actions, such as password reset, or disabling second-factor authentication.

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

We call G2V^{fpr}, G2V^{dis}, G2OT^{dis} and G2DT^{dis} the protocols 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 G2V^{fpr}, G2V^{fpr} and G2V^{dis}, G2OT^{fpr} and G2OT^{dis} and G2DT^{fpr} and G2DT^{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, G2OT^{fpr} and G2DT^{fpr}. Adding the action to the display would provide additional security guarantees.

Among the studied mechanisms, G2V^{dis} and G2DT^{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 G2V^{dis} performs better than G2DT^{dis} only in scenarios where we have $\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$, which may be considered as a powerful malware.

G2DT^{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

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		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✗✓✓	✖
			$\mathcal{M}^{dev}_{in:\mathcal{RO}}$	✖	✔
		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		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✓✓✓	✓✗✗
		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

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 which is *Google 2-step*, 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 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 its own url.

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

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 ($\mathcal{M}^{\mathsf{t-tls}}_{\mathsf{io}:\mathcal{RW}}$ or $\mathcal{M}^{\mathsf{u-tls}}_{\mathsf{io}:\mathcal{RW}}$), he may change the intended action and replace an untrusted login with a trusted one, thus providing the attacker with a trusted cookie. As a consequence, a login on an untrusted computer with U2F may enable future attacker logins on this computer.

This is a problem, as Yubikeys (an implementation of U2F token) claims protection against "phishing, session hijacking, man-in-the-middle, and malware attacks." While this holds for the first ones, this is not the case for the malware attacks. Moreover, one might expect that an external hardware token should allow users to login securely on an untrusted computer. Then, it is actually easy for an attacker to submit its own request to the user's token. Of course, a user has to press a button to accept a request. As we said previously, with a malware controlling the TLS connection, one press of the user may lead to many attacker logins, but this is a failure of the trust mechanism, not of U2F.

U2F may lead to another problem that is out of the scope of our analysis: the Yubikey does not have any way to provide feedback for a successful press. When the computer submits two request 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 similar to the problem identified during the analysis of the One-Tap mechanism: success and failure of the second factor should not be silent.

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. 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, success, 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. While cookies are a common mechanism, a new protocol called TOKENBINDING [21] is under development. 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. We describe the protocol in appendix in Figure 7. It may be seen as a partial U2F where the keys are directly stored on the computer.

We provide in Table 6 the results of the formal analysis of TOKENBINDING combined with U2F and $\text{G2DT}^{\mathsf{dis}}$. It provides protection against a read only access to the TLS interface, because it is not any more sufficient to steal the cookie. The attacker needs to be able to access the private key of the user which was generated on his platform but never sent over the network.

We remark that for TOKENBINDING, having full access to the computer interfaces is not enough to steal the secret key because it is generated directly on the computer and never sent over the network. We could have specified as we did for the others a memory interface and the corresponding malware. However, this malware scenario is not useful for our comparison : it would have lead to attacks for both the cookie and TOKENBINDING.

6. Pratical 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.

Short integer for $\text{G2DT}^{\mathsf{dis}}$. Currently, $\text{G2DT}^{\mathsf{dis}}$ uses only a 2-digit integer. Hence, an attacker has probability $1/100$ to guess the integer, which is much higher than usually accepted. To maintain usability 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 proposed ones.

Independence of the second factor. When trying to login on a malicious computer, we saw that the U2F token might be used by the attacker if he controls the USB interface. Therefore, our phone, which remains completely independent from the untrusted computer, is not affected by malware on the untrusted computer.

On the need of feedback. An advantage of the phone over the U2F token is also that the token does not provide feedback to the user, and two consecutive button presses may remain unnoticed. On the phone, a success or failure confirmation after pressing the button is easily provided to the user.

Storing the keys on a dedicated secure token. An advantage of the U2F token is that if your computer is compromised, the number of attacker logins is limited by the number of times the button is pressed. It is then more secure to store the keys on a device that ensures that keys are not completely compromised, rather than storing the keys on a computer or a smartphone, where a malware may extract them. We however saw that U2F does not provide perfect security, and if indeed the key cannot be compromised, one should be careful of how the token is used, to ensure that no unwanted computer becomes trusted, or that the user may not press twice the button in a row. A solution to mitigate key leakage for computer 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 question for 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.

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		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✓✗	✓✓✓	✖	✖
	FS		$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✓✗	✓✓✓	✓✗✗	✓✗✗
	FS	NC	$\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✓✓✗	✓✓✓	✖	✖

Table 6: Results for the TOKENBINDING extension

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.

Privacy. A concern about authentication that is often ignored is unlinkability. If the same second factor is used to login on different providers, e.g. the user's phone, it might be possible to use this second factor to link accounts. Unlinkability is obviously not provided by $G2DT^{dis}$, because one provides the same phone number. However, U2F explicitly claims to achieve unlinkability by generating a fresh key pair for every distinct provider.

7. 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 in with compromised agents. Moreover, we consider different levels of malwares, in a systematic way by representing a system as a set of interfaces. 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 a dialect of 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, which improves security.

As a direction for future work 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. $G2DT^{dis}$ 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.

Finally, as discussed above, U2F tokens claim to achieve unlinkability. It would be interesting to formally model and study this property.

Acknowledgements. We wish to thank Arnar Birgisson and Alexei Czeskis for interesting discussions and the anonymous reviewers for their comments. We are grateful for the support from the ERC under the EU's Horizon 2020 research and innovation program (grant agreements No 645865-SPOOC), as well as from the French National Research Agency (ANR) under the project Sequoia.

References

[1] P. A. Grassi, M. E. Garcia, and J. L. Fenton, "Nist special publication 800-63-3 – digital identity guidelines," Jun. 2017, available at https://doi.org/10.6028/NIST.SP.800-63-3.

[2] R. Morris and K. Thompson, "Password security: A case history," *Communications of the ACM*, vol. 22, no. 11, pp. 594–597, 1979.

[3] P. A. Grassi, J. L. Fenton, E. M. Newton, R. A. Perlner, A. R. Regenscheid, W. E. Burr, J. P. Richer, N. B. Lefkovitz, J. M. Danker, K. K. Choong, Yee-Yin Greene, and M. F. Theofanos, "Nist special publication 800-63b – digital identity guidelines – authentication and lifecycle management," Jun. 2017, available at https://doi.org/10.6028/NIST.SP.800-63b.

[4] "Google 2 Step," https://www.google.com/landing/2step/.

[5] "FIDO Yubikey," https://www.yubico.com/solutions/fido-u2f/.

[6] J. Bonneau, C. Herley, P. C. v. Oorschot, and F. Stajano, "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*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 553–567. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.44

[7] O. Pereira, F. Rochet, and C. Wiedling, "Formal Analysis of the Fido 1.x Protocol," in *The 10th International Symposium on Foundations & Practice of Security*, ser. Lecture Notes in Computer Science (LNCS). Springer, 10 2017.

[8] S. Kremer and R. Künnemann, "Automated analysis of security protocols with global state," *CoRR*, vol. abs/1403.1142, 2014. [Online]. Available: http://arxiv.org/abs/1403.1142

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

[10] A. Armando, R. Carbone, and L. Zanetti, *Formal Modeling and Automatic Security Analysis of Two-Factor and Two-Channel Authentication Protocols*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 728–734. [Online]. Available: https://doi.org/10.1007/978-3-642-38631-2_63

[11] D. Basin, S. Radomirovic, and L. Schmid, "Modeling human errors in security protocols," in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, June 2016, pp. 325–340.

[12] D. A. Basin, S. Radomirovic, and M. Schläpfer, "A complete characterization of secure human-server communication," in *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, 2015, pp. 199–213. [Online]. Available: https://doi.org/10.1109/CSF.2015.21

[13] "G Suite updates - Improved phone prompts for 2-Step Verification," https://gsuiteupdates.googleblog.com/2017/02/improved-phone-prompts-for-2-step.html.

[14] D. Dolev and A. Yao, "On the security of public key protocols," in *Proc. 22nd Symp. on Foundations of Computer Science (FOCS'81)*. IEEE Comp. Soc. Press, 1981, pp. 350–357.

[15] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *meltdownattack.com*, 2018. [Online]. Available: https://meltdownattack.com/meltdown.pdf

[16] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *meltdownattack.com*, 2018. [Online]. Available: https://spectreattack.com/spectre.pdf

[17] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," *SIGPLAN Not.*, vol. 36, no. 3, pp. 104–115, Jan. 2001. [Online]. Available: http://doi.acm.org/10.1145/373243.360213

[18] M. Abadi, B. Blanchet, and C. Fournet, "The applied pi calculus: Mobile values, new names, and secure communication," *Journal of the ACM*, vol. 65, no. 1, pp. 1:1–1:41, Oct. 2017.

[19] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and proverif," *Foundations and Trends® in Privacy and Security*, vol. 1, no. 1-2, pp. 1–135, 2016. [Online]. Available: http://dx.doi.org/10.1561/3300000004

[20] "Proverif source files," https://sites.google.com/site/multifactorformalverif/.

[21] A. Popov, M. Nystrom, D. Balfanz, A. Langley, N. Harper, and J. Hodges, "Token Binding over HTTP," Internet Engineering Task Force, Internet-Draft draft-ietf-tokbind-https-12, Jan. 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-tokbind-https-12

Appendix A. An attack on G2OT

We provide bellow the steps of an attack on G2OT when only the password is compromised:

1) The User enters its password and email.
2) Its browser tells him to press yes on his phone.
3) The attacker detects that the user contacted Google. After the first answer from Google the attacker blocks all further messages.
4) The Attacker initiates a session of its own by entering the user's email and password.
5) Depending on the timing, two things then happen:
   - The user press yes, nothing happens on its screen, and the attacker is logged in.
   - The user press yes, nothing happens on its screen, but another yes/no pops up on his phone. If he presses yes once more, the attacker is logged in.

In step 3, we block other answers from the server, because whenever two simultaneous login requests are submitted to the server, the server informs first one that there is a problem, resulting in an error message for the user. Thus to complete the attack in a completely invisible way, the attacker should be able to block the server response. If he cannot do this, depending on the timing, the user may not validate if he sees the error. Finally, in the attack description, we provide two cases, because we were not able to obtain a deterministic response from the phone. Indeed, when two login attempts where made at the same time, we sometime needed to validate twice on the screen, and sometimes only once. This may be due to the order of reception of requests, and session expiration on the server side. To be robust, the implementation should not accept this kind of situation, and 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. Indeed, we believe that it is plausible that users, after having pressed yes on their phone without any result, would press yes once more.

It is however important to mention that the attacker must be able to detect the user login, and block messages from the server to the client.

Appendix B. TokenBinding

TokenBinding is meant to be enabled on the trusted devices of the user, once the trust was established through a previously successful Multi Factor Authentication. We present it in Figure 7.

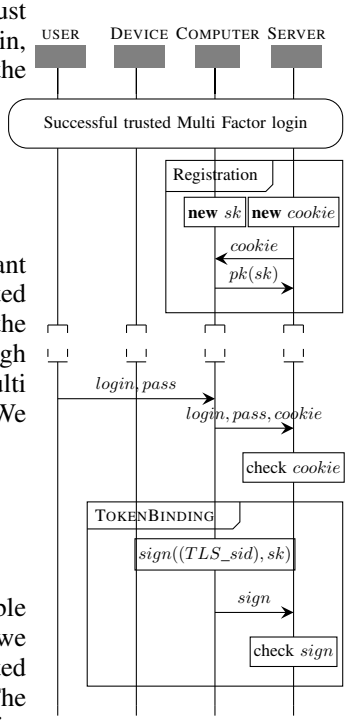

Figure 7: TokenBinding

Appendix C. Global results

We summarize in Table 7 and 8 all the results we computed using the automated generation of scenarios. 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.

Threat Scenarios	G2V	G2V^{fpr}	G2V^{dis}	G2C	G2OT^{fpr}	G2OT^{dis}	G2DT^{fpr}	G2DT^{dis}	U2F	U2F$_{tb}$	G2DT$^{dis}_{tb}$
	✔	✔	✔	✖	✔	✔	✔	✔	✔	✔	✔
PH	✖	✔	✓✓✓	✖	✔	✔	✔	✓✓✓	✔	✔	✓✓✓
NC	✔	✔	✔	✖	✖	✖	✔	✔	✔	✔	✔
FS	✔	✔	✔	✖	✖	✖	✔	✔	✔	✔	✔
PH NC	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✓	✔	✖
PH FS	✖	✖	✗✓✓	✖	✖	✖	✖	✗✓✓	✔	✔	✗✓✓
PH FS $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✗✓-	✓✓✗	✔	✗✓✓
PH FS $\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✗✓✓	✔	✔	✗✓✓
PH FS NC $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✔	✖
PH FS $\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✖	✖	✗✓✓	✖	✖	✖	✖	✖	✔	✔	✖
PH FS $\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
PH FS $\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✓✓✗	✖
PH FS $\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✔	✖
PH FS $\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$ $\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✔	✔	✖
PH FS $\mathcal{M}^{t-usb}_{io:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
$\mathcal{M}^{dev}_{in:\mathcal{RO}}$	✖	✖	✖	✖	✔	✔	✔	✔	- - -	- - -	✔
$\mathcal{M}^{t-dis}_{io:\mathcal{RO}}$	✔	✔	✔	✖	✓✓✓	✓✓✓	✓✓✓	✓✓✓	✔	✔	✓✓✓
$\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✓✓✓	✓✓✓	✖	✓✓✓	✓✓✓	✔	✔	✔	✔	✔
$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$	✖	✓✓✓	✓✓✓	✖	✓✓✓	✓✓✓	✔	✔	✔	✔	✔
NC $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✓✓✓	✓✓✓	✔	✔	✔
NC $\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✔	✔	✔	✔	✔
$\mathcal{M}^{dev}_{io:\mathcal{RO}}$	✖	✖	✖	✖	✔	✔	✔	✔	✖	✖	✔
$\mathcal{M}^{dev}_{in:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
$\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✖	✗✓✗	✓✓✗	✖	✗✓✗	✓✓✗	✗✓✗	✓✓✗	✖	✗✗-	✓✓✗
NC $\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✔	✔	✔	✖	✖	✖	✖	✖	✔	✔	✖
$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$	✖	✓✓✗	✓✓✗	✖	✓✓✗	✓✓✗	✓✓✗	✓✓✗	✓✓✗	✓✓✗	✓✓✗
NC $\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✗✗✗	✗✗✗	✖	✗✗-	✗✗✗
NC $\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✗✗✗	✗✗✗	✓✓✗	✓✓✗	✗✗✗
$\mathcal{M}^{t-dis}_{io:\mathcal{RO}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✖	✗✓✗	✓✓✗	✖	✗✓✗	✓✓✗	✗✓✗	✓✓✗	✖	✗✗✗	✓✓✗
$\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✖	✗✓✗	✓✓✗	✖	✗✓✗	✓✓✗	✗✓✗	✓✓✗	✖	✗✗✗	✓✓✗
NC $\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✓	✔	✖
NC $\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$ $\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✔	✔	✖
$\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✓✓✗	✓✓✗	✖	✓✓✗	✓✓✗	✓✓-	✓✓-	✓✓✗	✓✓✗	✓✓✗
$\mathcal{M}^{t-usb}_{io:\mathcal{RW}}$	✖	✓✓✗	✓✓✗	✖	✓✓✗	✓✓✗	✓✓✗	✓✓✗	✗✗✗	✗✗✗	✓✓✗
NC $\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
$\mathcal{M}^{t-usb}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✓✓✗	✓✓✗	✖	✓✓✗	✓✓✗	✓✓-	✓✓-	✖	✖	✓✓✗
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 $\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✔	✔	✔	✖	✖	✖	✖	✓✗✗	✔	✔	✓✗✗
FS $\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✖	✖	✓✗✗	✖	✖	✖	✗✗✗	✓✗✗	✖	✗✗✗	✓✗✗
FS $\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$	✖	✖	✓✗✗	✖	✖	✖	✗✗✗	✓✗✗	✓✓✗	✓✓✗	✓✗✗
FS NC $\mathcal{M}^{t-tls}_{io:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
FS $\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗✗	✓✓✗	✔	✓✗✗
FS $\mathcal{M}^{t-usb}_{in:\mathcal{RO}}$ $\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗✗	✔	✔	✓✗✗
FS NC $\mathcal{M}^{t-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✓✓✗	✔	✖
FS $\mathcal{M}^{t-usb}_{in:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗-	✓✓✗	✓✓✗	✓✗✗
FS $\mathcal{M}^{t-usb}_{io:\mathcal{RW}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗✗	✖	✖	✓✗✗
FS $\mathcal{M}^{t-usb}_{io:\mathcal{RW}}$ $\mathcal{M}^{t-tls}_{io:\mathcal{RO}}$	✖	✖	✓✗✗	✖	✖	✖	✖	✓✗-	✖	✖	✓✗✗

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
- G2DT$^{dis}_{tb}$- 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
- $\mathcal{M}^{interface}_{in:acc1,out:acc2}$- 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

Table 7: Global results for malware on trusted platform

Threat Scenarios		G2V	G2V^{fpr}	G2V^{dis}	G2C	G2OT^{fpr}	G2OT^{dis}	G2DT^{fpr}	G2DT^{dis}	U2F	U2F$_{tb}$	G2DT$^{dis}_{tb}$
	$\mathcal{M}^{u-dis}_{io:\mathcal{RO}}$	✔	✔	✔	✖	✔	✔	✔	✓✓✓	✔	✔	✓✓✓
	$\mathcal{M}^{u-tls}_{io:\mathcal{RO}}$	✖	✓✓✓	✔	✖	✓✓✓	✔	✔	✔	✔	✔	✔
	$\mathcal{M}^{u-usb}_{in:\mathcal{RO}}$	✖	✓✓✓	✔	✖	✓✓✓	✔	✔	✔	✔	✔	✔
NC	$\mathcal{M}^{u-tls}_{io:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✔	✔	✔	✔	✔
NC	$\mathcal{M}^{u-usb}_{in:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✔	✔	✔	✔	✔
	$\mathcal{M}^{u-tls}_{io:\mathcal{RW}}$	✖	✓✗✗	✓✓✓	✖	✓✗✗	✔	✓✗✗	✓✓✓	✖	✖	✓✓✓
NC	$\mathcal{M}^{u-dis}_{io:\mathcal{RW}}$	✔	✔	✔	✖	✖	✖	✖	✖	✔	✔	✖
	$\mathcal{M}^{u-usb}_{in:\mathcal{RW}}$	✖	✓✗✓	✔	✖	✓✗✓	✔	✓✗✓	✔	✓✗✓	✓✗✓	✔
NC	$\mathcal{M}^{u-tls}_{io:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✗✗✗	✖	✖	✖	✖
NC	$\mathcal{M}^{u-usb}_{in:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✗✗✗	✖	✓✗✓	✓✗✓	✖
NC	$\mathcal{M}^{u-dis}_{io:\mathcal{RW}}$ $\mathcal{M}^{u-tls}_{io:\mathcal{RO}}$	✖	✖	✖	✖	✖	✖	✖	✖	✔	✔	✖
NC	$\mathcal{M}^{u-usb}_{in:\mathcal{RO}}$ $\mathcal{M}^{u-dis}_{io:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✔	✔	✖
	$\mathcal{M}^{u-usb}_{io:\mathcal{RW}}$	✖	✓✗✓	✓✓✓	✖	✓✗✓	✔	✓✗✓	✓✓✓	✗✗✗	✗✗✗	✓✓✓
NC	$\mathcal{M}^{u-usb}_{io:\mathcal{RW}}$	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
FS	$\mathcal{M}^{u-tls}_{io:\mathcal{RO}}$	✖	✖	✗✓✓	✖	✖	✖	✔	✓✓✓	✔	✔	✓✓✓
FS	$\mathcal{M}^{u-usb}_{in:\mathcal{RO}}$	✖	✖	✗✓✓	✖	✖	✖	✔	✓✓✓	✔	✔	✓✓✓
FS	$\mathcal{M}^{u-dis}_{io:\mathcal{RW}}$	✔	✔	✓✓✓	✖	✖	✖	✖	✗✓✓	✔	✔	✗✓✓
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	$\mathcal{M}^{u-usb}_{in:\mathcal{RO}}$ $\mathcal{M}^{u-dis}_{io:\mathcal{RW}}$	✖	✖	✗✓✓	✖	✖	✖	✖	✗✓✓	✔	✔	✗✓✓
FS	$\mathcal{M}^{u-usb}_{io:\mathcal{RW}}$	✖	✖	✗✓✓	✖	✖	✖	✖	✗✓✓	✖	✖	✗✓✓

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
- G2DT$^{dis}_{tb}$- 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
- $\mathcal{M}^{interface}_{in:acc1,out:acc2}$- 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

Table 8: Global results for malwares on untrusted platform