TEnK-U: Terrorist Attacks for Fake Exposure Notifications in Contact Tracing Systems

Gennaro Avitabile[1], Daniele Friolo[1], and Ivan Visconti[1]

[1] *DIEM, University of Salerno, Italy*

Abstract

In this work we show that an adversary can leverage blockchain technology to attack the integrity of contact tracing systems based on Google-Apple Exposure Notifications (GAEN). We design a suite of smart contracts named TEnK-U allowing an on-line market where infected individuals interested in monetizing their status will then upload to the servers of the GAEN-based systems some keys (i.e., TEKs) chosen by an adversary. As a consequence, there will be fake exposure notifications of at-risk contacts arbitrarily decided by the adversary and allowed by infected individuals looking for money.

Such vulnerability can be exploited to anonymously and digitally trade valuable contact tracing data without a mediator and without risks of being cheated. This makes infected individuals prone to get bribed by adversaries willing to compromise the integrity of the contact tracing system for any malicious purpose. For instance, large-scale attacks with catastrophic consequences (e.g., jeopardizing the health system, compromising the result of elections) are easy to mount and attacks to specific targets are completely straight-forward (e.g., schools, shops, hotels, factories).

We show as main contribution a smart contract with two collateral deposits that works, in general, on GAEN-based systems and concretely with Immuni and SwissCovid. In addition, we show smart contracts with one collateral deposit that work with SwissCovid. Finally, we also suggest the design of a more sophisticated smart contract that could potentially be used to attack GAEN-based system even in case those systems are *repaired* to make the previous attacks ineffective. This last smart contract crucially uses DECO to connect blockchains with TLS sessions.

Our work shows that risks envisioned by Anderson and Vaudenay are absolutely concrete, in particular TEnK-U shows how to realize with Immuni and SwissCovid the terrorist attack to decentralized systems discussed by Vaudenay.

1 Introduction

During this COVID-19 pandemic several governments are deciding to use digital contact tracing systems in addition to other practices to contain the spread of SARS-CoV-2. The motivation is that digital contact tracing could help to notify at risk exposures to individuals that recently have been in close proximity to other individuals subsequently tested positive to SARS-CoV-2. If such systems worked perfectly, they would certainly be effective in alerting at risk individuals who, following some prescribed procedures (e.g., informing doctors, staying at home in self-quarantine), may significantly limit the spread of the virus. In some cases (e.g., in Switzerland), an alert received by a contact tracing smartphone application allows to get a test for free [Har20].

The most used contact tracing systems rely on Google-Apple Exposure Notifications (GAEN), a feature offered by recent updates of iOS and Android and therefore available on a large fraction of currently used smartphones. These systems are widely used in Europe (e.g., Austria,

Denmark, Estonia, Finland, Germany, Ireland, Italy, Latvia, Poland, Spain, Switzerland) and soon there will be cross-border compatibility [eN20].

Moreover, in the US, the states of Alabama and Virginia are using GAEN and several other states are planning to adopt such system soon. GAEN allows to run decentralized contact tracing where there is very low control from governments and therefore attacks from third parties are in general hard to mitigate.

Efficacy of digital contact tracing: the need of large adoption. Roughly, if n individuals out of m use the contact tracing app, the probability that a risky contact among two individuals is detected by the app is $(n/m)^2$. This very approximate estimation is anyway sufficient to understand that a truly beneficial help can come from such systems only when they are used by a large fraction of the population. Indeed, one should also take into account risky contacts with asymptomatic individuals, who clearly have less chances to be detected since only a small fraction of asymptomatic individuals is usually tested. Given such evident need for large adoption, many governments as well as Apple and Google are very actively promoting the use of national contact tracing apps.

Bluetooth low energy (BLE) and false positives due to technological limitations. The approach of GAEN-based contact tracing systems is to use BLE to detect close proximity contacts among smartphones. Each smartphone broadcasts a pseudonym via BLE, and this information is received by smartphones in close proximity along with some encrypted metadata. If a citizen is tested positive and decides to notify others, the receivers of her pseudonym will then manage to decrypt the stored metadata to then evaluate a risk factor[1]. Since BLE was not originally designed to detect a precise distance among devices, the evaluation of the risk factor is prone to significant errors. In particular, the fact that there can be obstacles (e.g., human bodies, clothes, purses) in between the two BLE devices negatively affects the precision of the distance estimation. As a consequence, in order to make sure that an alert can be generated when two individuals are close to each other (i.e., within 1.5/2 meters) despite obstacles, some important parameters have been increased in some countries [up20, oPH20c]. Therefore, even individuals that have been sufficiently far from each other (e.g., >2 meters) while properly using protective masks and always looking towards opposite directions, can be considered in close proximity and an alert is generated if one of them is then tested positive. The above issue, along with the fact that a notification does not specify time and location of the risky contact, indicates that when such systems are widely used, it is natural to expect false positive alerts(i.e., alerts received by users who have not been in contact with infected individuals). Depending of the efficiency of the health system of a country and the way it is connected to the digital contact tracing system, a high number of false positive alerts could overload the health system and heavily affect those individuals forced to a quarantine, even if healthy. Despite risks of false positive alerts, several governments keep promoting digital contact tracing, therefore considering the impact of those technological limitations marginal, or at least tolerable.

False positives due to attacks. While false positive due to BLE limitations in measuring distance can affect all individuals using the smartphone apps, a much more concerning threat allowing to direct false positive alerts to specific targets has been pointed out in prior work (e.g., see [Vau20a, Pie20, BCC$^+$20]). Indeed, GAEN-based contact tracing systems[2] can be heavily abused through replay attacks. Here the pseudonyms sent by an individual considered

[1] For example, metadata include information useful to estimate the distance among the smartphones which clearly impacts on estimating the risk of a contact.

[2] Sometimes for brevity we will just say GAEN systems.

at risk (e.g., showing evident symptoms) can be transmitted by an adversary to a different location in order to create a fake proximity contact. The attack can have a specific target but can also be performed at large scale. Recently, in [RGK20] Gennaro et al. discussed how the capability of running such attacks at large scale can be used to put a category of citizens in quarantine with the consequence of severely compromising the results of an election. In general, the malicious generation of false positives can be harmful in various ways, the health system can be overloaded of requests that can penalize those citizens who instead are really affected by the virus and would like to be tested as soon as possible, and preferably for free. A student can cause the complete closure of a school or university and similar attacks can be directed to shops, malls, gyms, post offices, restaurants.

Replay attacks were known already in April 2020 and in GAEN systems have a pretty large time window to be realized (about 2 hours). Nevertheless, governments have so far considered unlikely that such attacks can produce enough damage to cancel the positive effects of digital contact tracing. This is perhaps due to some complications involved in the attack. For example, an adversary may not want to get herself infected, or it could not be easy to identify, and be in physical proximity with, an individual who will be likely tested positive later on[3]. Still a replay attack is certainly not unlikely to happen and could contribute along with other attacks to produce serious damages (e.g., biasing the result of an election, affecting severely the quality of the selection process for testing individuals, damaging economy by stopping/delaying the activities of some strategic industries).

In [And20], Anderson conjectured the possibility of trolling attacks to spread panic and close schools. In [Vau20b], Vaudenay discussed the possibility of using smart contracts to realize a *terrorist* attack against decentralized systems, therefore the attack could potentially apply to GAEN-based systems as well. In this case, the attacker would spread on her targets some pseudonyms subsequently promising through a smart contract a reward to whoever uploads the corresponding keys. An infected individual that participates in this contract would therefore cash the reward and false positive alerts will raise on the smartphones of those targets decided by the terrorist.

More details about the above and various other attacks to GAEN systems are discussed in Section 1.2.

1.1 Our Contribution

In this paper we show that the terrorist attack conjectured by Vaudenay can be concretely mounted against currently deployed GAEN-based contact tracing systems. In particular, we have analyzed only two cases (i.e., Immuni [IT20] used in Italy and SwissCovid [oPH20a], used in Switzerland) and in both cases the terrorist attack can be successfully realized. We expect several other deployed GAEN systems to suffer from the same vulnerabilities.

More generally, our work shows how to attack the integrity of currently deployed GAEN-based contact tracing systems by leveraging blockchain technology. A very alarming side of our contribution is that current systems can be severely compromised without a need for the attacker to get infected and no need to be sure with high probability that an individual in close proximity to the attacker will be soon detected positive and will upload the keys. Our attacks use a suite of smart contracts named TEnK-U to establish a mediator-free market where parties, without knowing each other, without meeting in person and without running risks to be cheated, can abuse exposure notifications procedures of GAEN systems.

[3]Recall that a replay attack requires to start with pseudonyms of an individual that soon will be tested and detected positive and that will perform the upload procedure to allow the contact tracing system to notify a risky exposure.

As main contribution we show a generic smart contract Take-TEK that that works in general on GAEN systems and in particular with both Immuni and SwissCovid. Additionally we show two more generic smart contracts (Take-TEK-JWT, Auction-TEK) that require specific TEK-upload procedures and work with SwissCovid. All above smart contracts rely on servers of the contact tracing system publishing signed list of *infected* TEKs. Finally, we show that even without such a signed list of TEKs, an (certainly more sophisticated) attack is possible using DECO [ZMM+19]: we show another generic smart contract that relying also on off-chain protocols executed by seller and buyer can allow the smart contract to verify that the seller successfully uploaded TEKs during a TLS session with the contact tracing server.

All above smart contract are generic in the sense that we do not provide code directly runnable on a mainstream blockchain, but we anyway provide sufficient details so that providing runnable smart contracts merely consists of implementing our detailed specifications. From now on we will omit the word "generic" when referring to our smart contracts.

Trading TEKs exploiting publicly verifiable lists of *infected* TEKs. We show a smart contract named Take-TEK that allows a buyer (i.e., the adversary willing to spread false positive alerts) to decide the TEKs that will be uploaded by a seller (i.e., the infected individual that is willing to monetize his right to upload TEKs to the servers of the GAEN system). The smart contract requires the buyer to deposit the amount of cryptocurrency (we will call it prize) that she is willing to give to the seller. The seller instead will deposit an amount of cryptocurrency in order to reserve a time slot in which she will try to upload the TEKs. In case she will not manage to complete the upload of the TEKs, the deposit will be assigned to the terrorist. The deposit of the seller is therefore useful to make unlikely that a seller might try to prevent other sellers to complete the job. Additionally, we can hide the TEKs so that, even observing all transactions, it is not clear which TEKs have been traded among the many TEKs jointly published in a slot by the server of the contact tracing system. Take-TEK crucially relies on the server publishing such lists of TEKs along with a signature verifiable with a publicly known public key.

We show that the Take-TEK attack can be deployed to generate fake false positive alerts w.r.t. both Immuni and SwissCovid. Indeed, both systems follow strongly the design of GAEN and announce such signed lists of TEKs using ECDSA as signature scheme.

Regardless of Immuni and SwissCovid making or not available public keys of the signature scheme, we have successfully extracted the public keys from previously released signatures and therefore Take-TEK can be instantiated to attack both systems. More details are discussed in Section 2.

Trading TEKs exploiting publicly verifiable upload-authorization tokens. Despite being easy to carry out, the Take-TEK attack requires the seller to make a deposit which will remain blocked until the next update released by the contact tracing server. This, depending on the particular system, may take from a few hours up to one day. Removing the need for this deposit is therefore interesting from a theoretical point of view, and has also the practical effect of reducing the entry barrier, allowing more sellers to join this adversarial market. Interestingly, some systems (e.g., SwissCovid) provide infected users with long-lived upload authorization tokens that can act, for the seller, as a proof of having the right to upload TEKs. Such tokens include a signature computed with ECDSA, and thus it is possible to extract the corresponding public key by simply getting access to previously released tokens. Notice that tokens are not public, therefore a buyer will trust such smart contract only if he manages to access an already spent genuine token (e.g., if he received it in the past after being tested positive) released using the current public key.

Similarly to Take-TEK, we will show the design of a smart contract Take-TEK-JWT that includes the public key used to verify the validity of such tokens. The seller would then post a token (omitting its signature) and a non-interactive zero-knowledge (NIZK) proof of knowledge of a valid signature in order to reserve a time-slot to complete the upload procedure and cash the prize. Concretely, in Take-TEK-JWT part of the token along with a NIZK proof replaces the seller's deposit of Take-TEK. The attack works for the next-day JWT (i.e., the specific token used in this system to upload, on the following day, the TEK of the current day.) of SwissCovid (cfr., Section 3.3), but they may also work for other systems using the same idea. Moreover, we show the design of a smart contract Auction-TEK which exploits the credibility the token gives to the seller to allow him to be an auctioneer so that several buyers can compete, by placing bids, for having their TEKs uploaded to the official contact tracing server. We stress that in both cases the buyer needs to know the public key associated to the token. As discussed above, this knowledge is certainly possible but it does not come for free in SwissCovid. As a consequence, more effort is required for a buyer to participate in the attack. More details are given in Section 3.

Trading TEKs without publicly verifiable signatures: DECO. Given the above two cases, one might think that realizing the terrorist attack via smart contracts crucially relies on exploiting signed information which is produced within the contact tracing protocol itself. Indeed, up to now we relied on smart contracts knowing a special public key and using it to regulate the fate of deposited coins. Therefore, one might consider as a possible fix to such vulnerabilities to hide the public keys and to use a signature scheme such that it is hard to extract the public key from signed messages. However, we show that things are actually more complicated for designers of contact tracing systems. In particular, we show another way to buy/sell TEKs that follows a completely different approach. The key idea is requiring the seller to prove that a TLS session with the server led to a successful upload of the buyer's TEKs. The only requirements on the communication between smartphone app and server are that 1) both the TEKs and the positive (or negative) outcome of the upload procedure are part of the exchanged application data in the TLS session, and 2) the upload phase consists of just one request made by the client and the response of the sever (e.g., as it is in SwissCovid). At first sight, the attack seems very hard to realize since notoriously TLS produces deniable communication transcripts when it comes to application data (i.e., exchanged messages are only authenticated and not digitally signed). However, we exploit a very recent work of Zhang et al. [ZMM+19]. They show how to build a fully decentralized TLS oracle, named DECO, for commonly used ciphersuites.

Very roughly, two entities, a prover (the seller) and a verifier (the buyer), jointly perform the TLS handshake with the server. Let G be the generator of an Elliptic-Curve Group. Let $Y = s \cdot G$ be the message the client sends in the Elliptic-Curve version of the Diffie-Hellman (DH) key-exchange protocol. In DECO, the client's ($\mathcal{P}$ and $\mathcal{V}$ are seen by the server as a single client) contribution to the DH key is jointly computed by the prover and the verifier. In particular, the verifier sends a value (that we call a DH share) $Y_{\mathcal{V}} = s_{\mathcal{V}} \cdot G$ to the prover. The prover then will calculate her share $Y_{\mathcal{P}} = s_{\mathcal{P}} \cdot G$, and will compute $Y_{\mathcal{P}} + Y_{\mathcal{V}} = Y$, that is sent to the server (now $s_{\mathcal{P}} + s_{\mathcal{V}}$ corresponds to the DH secret s of the client). The server will then operate as usual by providing its DH message $Y_{\mathsf{S}} = s_{\mathsf{S}} \cdot G$ thus computing the shared DH key as $Z = s_{\mathsf{S}} \cdot Y$, from which it can derive the TLS session keys by evaluating a PRF. Given the DH contribute Y_{S} from the server, $\mathcal{P}$ and $\mathcal{V}$ respectively calculate their own secret shares $Z_{\mathcal{V}} = s_{\mathcal{V}} \cdot Y_{\mathsf{S}}$ and $Z_{\mathcal{P}} = s_{\mathcal{P}} \cdot Y_{\mathsf{S}}$ of the exchanged key Z (from which $Z_{\mathcal{V}} + Z_{\mathcal{P}} = Z$). The prover and the verifier will now perform a two-party computation protocol (2PC) starting from their own secret shares to derive shares of the MAC keys that will be used later to jointly compute the application

data messages (in the CBC-HMAC ciphersuite, after the 2PC, the entire encryption key is also obtained by the prover). When computing those TLS application data messages, since such messages require to compute a MAC, prover and verifier run a two-party computation protocol to compute the MAC while keeping their own shares of the MAC key private. The message on which the MAC is computed also remains private, and is not disclosed to the verifier. When the communication with the server ends, in order to preserve data confidentiality, the prover proves to the verifier statements on the exchanged application data using zero-knowledge proofs. For our purposes zero-knowledge proofs are not essential, therefore we will not deal with them and we will use DECO in a modified and greatly simplified form.

We keep running the DECO protocol off-chain but we find a way to connect the DECO run between the prover and the verifier to the state of the smart contract, so that the smart contract will eventually be able to act as an impartial judge ensuring both parties behaved honestly, or punishing the malicious one otherwise. We let the verifier create, before performing the TLS handshake, a smart contract with a deposit indicating the prize for the seller. The requested TEKs are also posted on the smart contract when it is initialized. To prevent the verifier from cheating, he is also supposed to fix, at the beginning, the secret $s_{\mathcal{V}}$ he is going to use during the DECO protocol run by providing his DH share $Y_{\mathcal{V}} = s_{\mathcal{V}} \cdot G$. After that, the prover will reserve a time slot in the smart contract during which the prover and the verifier will exchange messages off-chain. We slightly modify the DECO handshake phase by making the prover verify that the share $Y_{\mathcal{V}}$ provided by the verifier is indeed the one posted on the smart contract. If this is not the case, the prover aborts preventing the verifier from cheating afterwards. At the end of the off-chain communication, the prover will send, in clear, the application data messages exchanged with the server , along with their MAC tags, the server's DH message Y_{S}, and her own DH share $Y_{\mathcal{P}}$. The verifier then, will have to publish a valid preimage coherent with the previously posted DH share (i.e., $s'_{\mathcal{V}}$ such that $s'_{\mathcal{V}} \cdot G = Y_{\mathcal{V}}$). The smart contract will then locally re-calculate the MAC key, the MAC tags and pay the seller if everything verifies and the messages are as expected. Further details are described in Section 4.

Remark on the actual work done by our smart contracts. All of our smart contracts provide full guarantees to both seller and buyer at the expense of running some cryptographic operations that can obviously produce transaction costs. Nevertheless, we notice that the expensive computations might in practice happen very rarely if we make an additional optimization based on pragmatism. Indeed, we notice that the main computational cost for those smart contracts consists of checking at the very end that the seller has completed the task of uploading TEKs correctly. We observe that a buyer can check on its own that TEKs are published by the server, and can be happy that the trade has been completed successfully. Therefore, it is natural to expect that the buyer would give his approval to the smart contract to transfer the money to the seller avoiding the execution of expensive computations, and therefore saving transaction costs[4]. Since this behavior would be visible in the wild, the reputation of the buyer would also benefit of such approvals and more sellers would want to run contracts with him. Moreover, a (somewhat irrational) buyer that refuses to speed up the execution of the smart contract would anyway not stop the final transfer of the deposited money to the seller. As a result, the buyer would get a worse reputation. In turn, the expensive work done by our smart contracts belongs to pieces of code that would rarely be executed in practice. Pragmatically

[4]Obviously, the smart contract can be adjusted so that, in case the buyer does not give his approval and the seller shows that she completed successfully her part of the contract, the expensive transactions costs due to the lack of help of the buyer are charged to the wallet of the buyer. A simple way to realize this could be asking for a larger deposit made by the buyer which could clearly cover the transaction costs of the seller in case the buyer does not give his approval and seller shows that she successfully completed the upload procedure.

speaking, all our smart contracts even when adding stronger protection (e.g., hiding the traded TEKs) are very efficient when run in practice, except the last smart contract that without the approval of the buyer does not protect seller privacy and is furthermore quite expensive.

1.2 Related Work

The design of GAEN is very similar to the low-cost design of DP-3T [DT20], and thus several vulnerabilities identified in prior work generally apply to both systems. In [Vau20a], Vaudenay reports both privacy and security issues. The most famous privacy attack is the so-called *Paparazzi* attack. Basically, it is possible to track infected individuals over a certain time window[5] during which pseudonyms are linkable. All the attacker needs to do is to place passive antennas in the locations of his interest. A proof of concept implementation of the Paparazzi attack has been realized by Seiskari [Sei20]. Several limitations of GAEN systems are also discussed in [ABIV20].

Regarding security issues, Vaudenay extensively considers false alert injection attacks, where the adversary manages to to raise false alerts on the smartphone apps of targeted victims. Within this category, there are *replay* and *relay* attacks. Replay attacks involve capturing a pseudonym P at a location L_1 and transmitting it, at a later time, at a different location L_2. In relay attacks, instead, pseudonyms re-broadcasting happens in real time. If such attacks are successful, users may be alerted even if they have never met an infected person. GAEN is vulnerable to relay attacks and to replay attacks carried out within two hours [Goo20b]. Vaudenay in [Vau20a] and Pietrzak in [Pie20] proposed already in April 2020 some solutions to defeat these attacks, but they have not been included in DP-3T and GAEN designs so far. Baumgärtner et al. [BDF$^+$20] provide empirical evidence of the concrete feasibility of both Paparazzi and replay attacks. Pietrzak et al. [Aus20] analyze inverse-sibyl attacks in which multiple adversaries cooperate to use the same pseudonyms. If one of the attackers gets to upload his TEKs, many false alerts may be raised. This attack can be easy to realize to generate a large amount of false positive alerts both when mounting a replay attack and when mounting our attacks based on smart contracts.

Several GAEN-based systems are currently used in the world for digital contact tracing. Vaudenay and Vuagnoux, and later Dehaye and Reardon extensively evaluated SwissCovid [VV20b, VV20a, VV20c, DR20a, DR20b], confirming some vulnerabilities showed in previous works and elucidating new ones. For example, desynchronized rotation of pseudonyms and BLE MAC address makes certain phones traceable even w.r.t. citizens that have never been infected; this is known as *Little Thumb* attack [VV20d]. In addition, a bug of SwissCovid allowed anyone to upload arbitrary TEKs by just setting the JWT algorithm field to null[6].

Finally, another class of attacks leading to false alerts involves bribing. Vaudenay envisions various possibilities for the development of dark economies [Vau20b] which could support false alert injection attacks, allowing them to be carried out at very large scales. In particular, the *Lazy Student* attack describes a dark economy where hunters collect pseudonyms of individuals who will likely become infected later on, and deposit them on a smart contract. If such pseudonyms are uploaded to the contact tracing system, the hunter gets a reward paid by a buyer. If replay attacks are doable, the buyer of such pseudonyms (i.e., the lazy student) can use them to make target victims' apps raise false alerts. As noted by Vaudenay, this dark economy is sustainable only if the smart contract has a way to check that pseudonyms were actually

[5] In GAEN this time amounts to 14 days if the adversary colludes with the authorities, and otherwise to one day assuming TEKs are properly mixed and anonymized prior to publication.

[6] The bug was found and responsibly disclosed by André Cirne. More on this at `https://mrsuicideparrot.github.io/security/2020/07/30/CVE-2020-15957.html`and `https://www.melani.admin.ch/melani/en/home/public-security-test/current_findings.html`

reported to the official server. Another form of dark economy described by Vaudenay is the *terrorist* attack. It involves users reporting pseudonyms that differ from the ones used during previous days. In fact, in both Immuni and SwissCovid there is no mechanism enforcing users to upload genuine TEKs. Again, a TEK could be posted on a smart contract automatically issuing a reward to whoever reports it to the contact tracing system. This purchase may lead to a massive amount of fake notifications, without relying on replay attacks that instead is required by the Lazy Student attack. The huge impact of such attacks seems to have gone unnoticed or just ignored. In [LHML20] the cybersecurity risks of contact tracing systems are reviewed and compared using a subjective scoring scheme. The report considers injection of false alerts notifications by only mentioning replay attacks or trivial attacks such as recruiting people with symptoms. The terrorist attack is not even mentioned and thus, in light of our results, their scores might require some adjustments.

Vaudenay and Vuagnoux expressed these and other concerns in their analysis of SwissCovid [VV20a]. The Swiss National Cyber Security Center (NCSC) answered to their criticism seemingly downplaying those risks. The possible development of dark economies was ignored [Cen20a] and a recap table on security issues reports on SwissCovid marks the concerns expressed by Vaudenay as addressed, including false alert injection attacks (see page 8 [Cen20b]). Nevertheless, no solution or mitigation to such problems is reported. In this paper, we show how GAEN's design is prone to abuses allowing smart contracts to automatically verify that some given TEKs have been uploaded to the backend server. This demonstrates that harmful scenarios envisioned by Vaudenay can actually be reality in those countries using GAEN-based contact tracing systems. Such dark economies could act as a platform facilitating certain attacks that could jeopardize the health system of a country, slow down target businesses or even compromise the result of elections as recently discussed in [RGK20].

Bribing attacks on smart contracts. As we will discuss in more details in Section 2, trading TEKs is practically realizable and reduces the risk of one party of interacting with a dangerous entity like a criminal, at the minimum. Bribery attacks on Smart Contracts for different scenarios have been proposed in the context on bribing miners in Ethereum and Bitcoin [MHM18, LK17, TJS16, VTL17, KNW20].

2 Trading TEKs in GAEN Systems

The GAEN API has been created to provide an efficient platform for exposure notifications on top of which countries can easily develop digital contact tracing systems. GAEN is supposed to solve various technical problems (e.g., changing BLE MAC address synchronously with the rotation of pseudonyms, keeping BLE advertisements on in background) on a large fraction of available smartphones[7]. At the same time, Google and Apple put themselves in the role of regulators creating a somewhat inflexible API that basically forces anyone who is willing to benefit from it to adopt a specific design for contact tracing. What is left in the hand of the developers is merely the creation of the graphical interface, the choice of some parameters and the realization of a server to gather and spread data about infected users and, more importantly, an authentication mechanism to avoid the upload of data by non-infected users.

A useful overview of the above state of affairs, along with privacy and security issues of GAEN and SwissCovid (i.e. the Swiss contact tracing application), has been reported by Vaudenay and Vuagnoux [VV20b].

[7] Indeed, see the case of UK that tried to develop a system without GAEN but had to give up because of technical barriers `https://www.bbc.com/news/technology-53095336`.

```
start_timestamp: 1591254000 //start of the time window of included keys
end_timestamp: 1591268399 //end of the time window of included keys.
region: "222"
batch_num: 1
batch_size: 1
signature_infos {
       verification_key_version: "v1" //version of the used verification key
       verification_key_id: "222"
       signature_algorithm: "1.2.840.10045.4.3.2"
       1: "it.ministerodellasalute.immuni"
}
keys {
       key_data: ".." //base64 encoded TEK
       transmission_risk_level: 8
       rolling_start_interval_number: 2651616 //date of usage of TEK
       rolling_period: 144
}...
```

Figure 1: Example of an `export.bin` file for Immuni, the Italian contact tracing app. The meaning of the key fields is commented on the side. The `start_timestamp` and `end_timestamp` are expressed in UTC seconds, `rolling_start_interval_number` is expressed in 10 minutes increments from UNIX epoch. The `export.sig` contains the digital signature of the `export.bin` file, along with the field `signature_infos`.

In order to evaluate the contagion risk, GAEN provides an appropriate method. In the android version this method is called `provideDiagnosisKeys`, while in iOS it is called `detectExposures` [Goo20e, App20a]. Such two methods require various arguments and among them there must be two files with a specific format. The first file is named `export.bin` and contains, along with other fields, a list of TEKs belonging to infected users that have decided to perform the upload procedure. Each TEK has also a date attached, which indicates when such TEK was used. The second file, named `export.sig`, contains a digital signature of the file `export.bin` [Goo20a, App20b]. An example of `export.bin` is shown in Figure 1. The methods `provideDiagnosisKeys` and `detectExposures` match the reported TEKs with the smartphone's internal database if and only if the digital signature verifies under a public key that has been previously communicated by the developers to Apple and Google. Google motivates this requirement saying that it ensures that keys received by the devices are actually from the authorized server and not from malicious third parties [Goo20c]. However, this is also automatically accomplished by using TLS while connecting to the server. As many contact tracing systems use Content Delivery Networks (CDNs) to disseminate TEKs (e.g., the CDN used by Immuni is operated by Akamai, while the one used by SwissCovid by Amazon), perhaps this requirement has been put in place to thwart malicious modifications operated by the CDN itself. Unfortunately, as we will see next, this requirement paves the way for the development of dark economies where TEKs to be uploaded by infected users are traded through smart contracts.

2.1 Take-TEK Smart Contract: Buying/Selling TEKs Uploads

Whenever a user is tested positive, she is given the right to upload her TEKs to the server so that the other users can be notified a risk of infection. The mechanism can be implemented in different ways. For example, a simple method consists of a code generated by the app that is given first to the health operator in order to activate it on the server. Then, once the server has authorized the code, the app will upload the TEKs along with the code (e.g., Immuni follows this approach). More complex mechanisms may be put in place. However, the attack we show next works for every GAEN-based contact tracing system under some natural assumptions that we will discuss later.

To simplify the description, from now on we talk about `export.bin` as a list of pairs of values. In each pair the first value is a TEK and the second value is the corresponding date of usage date (i.e., the `rolling_start_interval_number`). Let the seller $\mathcal{P}$ be an infected user who would like to monetize her right to upload TEKs, and buyer $\mathcal{B}$ someone who is interested in paying $\mathcal{P}$ in order to upload TEKs of her interest. If the seller is able to prove she acted as promised, this selling process can be executed remotely remaining automated, anonymous, and scalable. GAEN's choice for the list of TEKs to be signed by the server makes the verification easy to the smart contract, therefore it greatly facilitates such trades.

To be more specific, the trade can be performed using a blockchain capable of executing sufficiently powerful smart contracts (e.g., Ethereum). Such smart contract guarantees that $\mathcal{P}$ gets an economic compensation if and only if $\mathcal{P}$ uploads to the server the TEKs specified by $\mathcal{B}$. In fact, the requirement of having a digital signature makes possible for the smart contract to easily verify that $\mathcal{P}$ held up his end of the bargain and therefore can be rewarded. Implementing this kind of check would not be so straightforward otherwise since smart contracts cannot execute non-deterministic calls such as checking on the internet for updates.

The high-level functioning of the smart contract is as follows. $\mathcal{B}$ creates the smart-contract posting a list of TEKs with the related date, and deposits a prize to be redeemed by a seller. An interested $\mathcal{P}$ also makes a small deposit to declare her intention to upload the TEKs specified by $\mathcal{B}$ (the purpose of this small deposit is explained later). After having made this deposit, $\mathcal{P}$ has a specified amount of time to complete the upload procedure. Before the time runs out, $\mathcal{P}$ must provide a list of TEKs which includes all the pairs $(\mathsf{tek}, \mathsf{date})$ specified by $\mathcal{B}$, along with a valid signature under the server's public key. If $\mathcal{P}$ manages to do so, she gets a reward, otherwise both deposits go back to $\mathcal{B}$. By making a deposit, the seller reserves a time slot during which she can perform the upload. Such deposit protects the buyer from denial of service (DoS) attacks by sellers who actually do not have the right to upload TEKs. Here, as in the remainder of the paper, with the word DoS we mean attacks carried out by fake sellers which prevent honest sellers from participating to the trade.

We name the above smart contract Take-TEK and the attack that leverages the use of this smart contract Take-TEK attack.

The time window given to $\mathcal{P}$ must be wide enough to take into account that new TEKs are not continuously released by the server, in fact, several hours may pass between the submission of a TEK and its publication. Obviously the amounts of both deposits will be significantly higher than transaction fees. A custom software is needed to upload arbitrary TEKs, however, this simple software may be developed even by other entities (not just the buyers), and publicly distributed on the Internet or other sources (e.g., Darknet). Therefore, all the seller would need to do is just downloading and using the software that could run both on smartphones or on computers, being easily approachable by a large fraction of the infected citizens willing to gain money[8].

[8]COVID-19 by itself caused a global economic crisis which led to lower wages and job losses. More details at

Additionally, the time given to the seller to complete the upload after having been tested positive must be long enough to reserve a slot on the blockchain (i.e., enough to wait that the transaction related to the seller's deposit gets confirmed) and subsequently send the TEKs via the custom software.

Various proposed upload authorization mechanisms include manual steps (e.g., SwissCovid uses an authorization code, termed covidcode, which lasts for 24 hours [oPH20b]) which, in order to function properly, naturally give the seller enough time to perform the steps mentioned above. For example, if a code is communicated to the infected user via a phone call, he should be given a fairly large amount of time to write down the code and insert it in the app later on (the needs of people with disabilities and of elder people must be taken into account). Even systems that have fairly strict requirements on the time by which the upload procedure must be completed since the authorization token is given, should allow for errors and recovery procedures, which may give additional time to the future seller. For example, Immuni[9] requires that the infected user dictates, via phone call, a code that appears on his device. After that, the user must complete the upload within two minutes. If this does not happen, the procedure must be repeated. Additionally, the system should be tolerant. People should have the opportunity to perform the upload procedure later on if they are unable to do it in that precise moment. For example, consider the case of someone who is not at home while receiving the call and, after having heard of such bad news, wants to complete the operation in the comfort of her place (not all people who take a test have to stay in self-isolation). It is worth noting that strict requirements on the upload phase reduce user's privacy. A clear example of this is the just described mechanism of Immuni, by which the medical operator, by checking whether a code has been used or is instead expired, gets to know whether or not the infected user actually uploaded her TEKs. Anyway, we would point out that, as things stand at the moment, due to the huge connection timeout (2 hours) of the Immuni (and SwissCovid) key servers (see Appendix D for more details), sellers do not necessarily need to ask to the health authority to repeat the uploading procedure after reserving a slot in the blockchain. Right after receiving the code, a seller can start the TLS handshake with the server, keep the connection open, send the transaction to the blockchain (thus waiting for confirmation), and complete the uploading procedure on time.

Take-TEK: attack description. Let us consider a buyer $\mathcal{B}$ who wants to pay an infected person, that we call seller $\mathcal{P}$, to upload $\mathcal{B}$'s TEKs to the contact tracing system. $\mathcal{B}$ and $\mathcal{P}$ owns wallets $\mathsf{pk}_{\mathcal{B}}$ and $\mathsf{pk}_{\mathcal{P}}$ respectively. In this setting the buyer has no assurance that the seller is actually an infected person, and she is not just a malicious party trying to slow down the buyer's plan. Thus, some collateral must be deposited from $\mathcal{P}$ too. The seller will lose the collateral in case she is not able to prove that she sent the buyer's TEKs to the server S. We assume the existence of an unforgeable signature scheme $(\texttt{Gen}_{\mathsf{S}}, \texttt{Sign}_{\mathsf{S}}, \texttt{Ver}_{\mathsf{S}})$ used by the server. A brief overview of the main functions follows below.

$\texttt{Constructor}(\mathbf{T}_{\mathcal{B}}, \mathsf{vk}_{\mathsf{S}}, t, d_{\mathcal{P}})$: It takes as input a set of tuples $\mathbf{T}_{\mathcal{B}} := (\mathsf{tek}_i^{\mathcal{B}}, \mathsf{date}_i^{\mathcal{B}})_{i\in[n]}$ with $n \leq \mathsf{maxteks}$ [10], where tek_i is the i-th TEK of the buyer and date_i is the associated date, the verification key vk_{S} to be used to verify the signature of the TEKs list, a timestamp

https://en.wikipedia.org/wiki/COVID-19_recession.

[9]See page 9 of this presentation held by an Italian Health Authority representative (the reference is in Italian). http://www.amcli.it/wp-content/uploads/2020/07/7-luglio-2020-BATTILOMO-MINSALUTE-App-Immuni.pdf

[10]The maximum number of TEKs that can be uploaded in one shot depends on the particular application. GAEN provides up to 14 TEKs related to the previous two weeks. SwissCovid, allows to upload only keys between the onset of symptoms and the upload date. Obviously, if the smart contract demands more than $\mathsf{maxteks}$ TEKs, no one would ever be interested in the transaction.

t, indicating the maximum time the seller has to provide the correct list and signature, and the collateral value $d_{\mathcal{P}}$ that the seller must deposit.

`Deposit()`: must be triggered by $\mathcal{B}$ and takes as input a quantity p of coins as the payment for the seller.

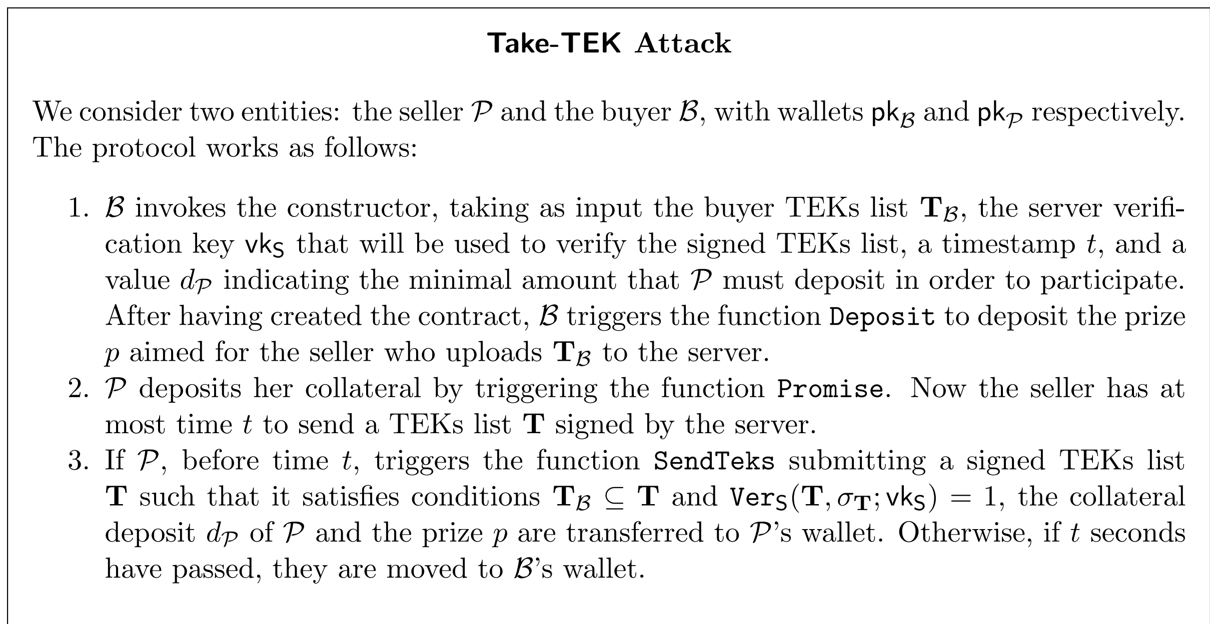
`Promise()`: can be triggered by the seller $\mathcal{P}$ by sending a quantity of collateral deposit $d_{\mathcal{P}}$ as a payment when invoked.

`SendTeks`$(\mathbf{T}_{\mathsf{KS}}, \sigma_T)$: can be triggered by the seller $\mathcal{P}$ to provide a list of TEKs together with its signature σ_T. Let the list released by the server be $\mathbf{T} = (\mathsf{tek}_i, \mathsf{date}_i)_{i \in [N]}$, where N is the number of published TEKs. It checks that:

- $\mathbf{T}_{\mathcal{B}} \subseteq \mathbf{T}$,
- $\mathtt{Ver}_{\mathsf{S}}(\mathbf{T}, \sigma_{\mathbf{T}}; \mathsf{vk}_{\mathsf{S}}) = 1$.

If both checks pass, $d_{\mathcal{B}}$ coins are transferred to the seller's wallet $\mathsf{pk}_{\mathcal{P}}$.

The protocol description is depicted in Figure 2.



Take-TEK Attack

We consider two entities: the seller $\mathcal{P}$ and the buyer $\mathcal{B}$, with wallets $\mathsf{pk}_{\mathcal{B}}$ and $\mathsf{pk}_{\mathcal{P}}$ respectively. The protocol works as follows:

1. $\mathcal{B}$ invokes the constructor, taking as input the buyer TEKs list $\mathbf{T}_{\mathcal{B}}$, the server verification key vk_{S} that will be used to verify the signed TEKs list, a timestamp t, and a value $d_{\mathcal{P}}$ indicating the minimal amount that $\mathcal{P}$ must deposit in order to participate. After having created the contract, $\mathcal{B}$ triggers the function `Deposit` to deposit the prize p aimed for the seller who uploads $\mathbf{T}_{\mathcal{B}}$ to the server.
2. $\mathcal{P}$ deposits her collateral by triggering the function `Promise`. Now the seller has at most time t to send a TEKs list $\mathbf{T}$ signed by the server.
3. If $\mathcal{P}$, before time t, triggers the function `SendTeks` submitting a signed TEKs list $\mathbf{T}$ such that it satisfies conditions $\mathbf{T}_{\mathcal{B}} \subseteq \mathbf{T}$ and $\mathtt{Ver}_{\mathsf{S}}(\mathbf{T}, \sigma_{\mathbf{T}}; \mathsf{vk}_{\mathsf{S}}) = 1$, the collateral deposit $d_{\mathcal{P}}$ of $\mathcal{P}$ and the prize p are transferred to $\mathcal{P}$'s wallet. Otherwise, if t seconds have passed, they are moved to $\mathcal{B}$'s wallet.

Figure 2: Interaction between the buyer $\mathcal{B}$ and the seller $\mathcal{P}$ to carry out the double deposit bribery attack.

2.2 Subtleties in the Wild

In Section 2.1 we gave an high-level overview of how uploads can be sold safely via blockchains. However, there are some subtleties we overlooked for the sake of simplicity. In this section, we first analyze the advantages for adversaries when using automated trade compared to already known attacks. Then we consider certain problems that arise while trying to concretely mount our attack against deployed GAEN-based contact tracing systems. We also show how these difficulties are easily tackled if very small modifications to our attack are made.

Advantages (for an adversary) of automated trade. One might think that malicious injection of fake TEKs is inherent in decentralized contact tracing systems since there is no control over the smartphone of an infected person and thus, when the time of the upload comes,

the infected person can always use a smartphone belonging to someone else. Moreover, even in case some digital procedure with the smartphone is required when the individual is tested, still a different smartphone could be used.

While it is true that such simple attacks are very hard to tackle, they have limited impact for at least two main reasons: 1) the buyer must handover his smartphone to the seller, and this requires physical proximity; 2) sellers and buyers must trust each other since an illegal payment must be performed without being able to rely on justice in case of missing payment or aborted upload of keys. Indeed, even if in need of money, people are generally afraid of dealing with criminals since they may get scammed or threatened. Additionally, the buyer might expose the sellers' identities to the authorities in case he gets arrested or legally persecuted. Equally, the buyer may share the same concern with respect to an unreliable seller. It goes without saying that some citizens are prone to violate the rules[11] when they believe that risks are low compared to the advantages.

As such, the simple and inherent attacks consisting of exchanging smartphones, or even of using a malicious app with TEKs sent by a criminal contacted directly by the infected citizen, do not scale and can pollute the system in some *tolerable* way.

Having a mechanism which allows this trade to happen remotely, in anonymity and ensuring no party is cheated, solves all the above problems for parties willing to abuse of such contact tracing systems. In fact, it provides a framework for large-scale black markets of TEKs. The seller would not feel threatened in any way and could easily earn money, on the other hand, the buyers would benefit from a larger sets of users to be in business with, therefore succeeding in many possible attack scenarios.

Extracting public keys from signatures. Take-TEK obviously requires that the server's public key is known to both the involved parties. This guarantees that the buyer is sure the reward is paid only to sellers who actually upload data to the contact tracing system, and that honest sellers are sure they will be able to satisfy the conditions to be paid, namely obtaining a valid digital signature for reward redemption. A Github issue asking for the public key of the Italian contact tracing app has been opened on the 7th of June and it has still not been addressed at the time of writing [um20]. SwissCovid Android app contains a bucket public key value inside the configuration file (the values `BUCKET_PUBLIC_KEY` and `CONFIG_CERTIFICATE` can be found in [ST20b]) that is used to perform signature verification outside GAEN. Anyway, as we can notice with Immuni, this is not a requirement. One might think that keeping secret verification keys may prevent attacks as the one of Section 2.1. However, it turns out that it is actually not the case. In fact, since GAEN uses ECDSA, starting from a signature and the related message we can recover two candidate public keys, one of which will match the actual one with overwhelming probability. A practical example showing this procedure can be found in [Yan19]. Such message/signature pairs are generally made publicly available and are easily accessible by appropriately querying the server of the specific contact tracing system. Multiple pairs per day may be released. A comprehensive description on how to get this data has been provided by the Testing Apps for COVID-19 Tracing (TACT) project, along with scripts to automate the downloading process [LF20]. We also practically performed the extraction procedure, successfully extracting the keys for both SwissCovid and Immuni which, as every GAEN-compliant contact tracing system, use ECDSA to sign TEKs to be distributed for contagion risk evaluation.

[11] Note that the infected person is also committing a violation by allowing the injection of external TEKs.

Updates of public keys. There is a subtle technical problem with the attack described in Section 2.1. The digital signature keys that the server uses may change over time. In fact, as shown in Figure 1, the `export.bin` file includes a field indicating a version for the verification key. This field follows a progressive numeration, that is the first version is termed v1, the second one v2 and so on. This means that the server may change the verification key it uses, perhaps within a set of keys that have been pre-shared with Google and Apple. Therefore, it might happen that, after the seller makes the deposit and accepts to upload the buyer's TEK, the server, by coincidence, decides to use a new key which was never used before, thus producing a signature that is not verifiable under the public key posted on the smart contract.

However, by making a slight modification to the smart contract, it is possible to handle also this unfortunate event. Having realized that he would be unable to redeem the reward, the seller might activate a special recovery condition. After this, the buyer will be able to collect both deposits if and only if he manages to provide a pair of export files which have an `end_timestamp` (see Figure 2) subsequent to the time of the recovery request and verify under the public key originally posted on the smart contract; otherwise the deposits are returned to the original owners. Obviously, enough time should be given to the buyer to provide the export files, similarly to what happens to the seller after her deposit.

This event is certainly very annoying for the seller and might play as disincentive to join the trade, but taking a look at real-world data one realizes that this is a relatively rare event. We considered several countries which are currently using a digital contact tracing system, namely: Italy, Switzerland, Austria, Germany, Ireland, Northern Ireland, Denmark, Latvia, Canada and US Virginia. Until August 23rd (last time we checked), only US Virginia and Italy have switched to the second version of the verification key. In particular, the change to the Italian system dates back to the 15th of June [12] and no modifications have been made since then. Notably, some countries' systems, like Switzerland and Germany's ones, are active from several months now and the verification key has not changed at all. To the best of our knowledge, the criteria by which the verification key should change is not documented anywhere.

A worry-free seller. As previously discussed, the effectiveness of a digital contract tracing system is strictly related to various factors among which the percentage of active population using them. Appropriate measures should be taken to earn citizens' trust since it is the only way to guarantee broad adoption. With this in mind, the European Commission released a series of recommendations in relation to data protection stating the need of identifying solutions that are the least intrusive and comply with the principle of data minimization [Com20]. A similar recommendation has been given by the Chaos Computer Club (CCC) [Clu20], the Europe's largest ethical hackers association, which explicitly states that "data which is no longer needed must be deleted". Corona-Warn, the German contact-tracing system, declares to be fully compliant with CCC's guidelines [CT20b]. Many other systems are inspired by similar principles. For example, the Italian system Immuni also declares that data is deleted when no longer needed [IT20], as well as the Swiss system SwissCovid which also specifies a retention period for the TEKs and the upload authorization codes [oPH20b]. In its recommendation to build a verification server authenticating the uploaded TEKs, Google states that identifiable information should not be associated with uploaded data [Goo20d]. Corona-Warn's documentation also clearly specifies that all upload authorization data are retained only as far as they are needed for the authenticating uploads [CT20a].

The adoption of the above measures ensures that uploaded data do not link to, nor identify a particular individual. This is an important contract between government and citizens, therefore, it should be impossible for a government to link the TEKs with the real identity of users who

[12] This change occurred in the 4th export file.

submitted them without violating the contract with citizens, that, at least in part, would obviously loose trust in the government, delete the apps, switch their preference towards other political parties and so on.

This is very important considering that GAEN systems are vulnerable to the Paparazzi Attack described by Vaudenay [Vau20a]. Basically, it is possible to link an infected users' pseudonyms tracing him over the duration of a TEK, or for 14 days if the TEKs are linked, by only using passive antennas. Leaving to malicious parties the ability to link this data to a person's real identity would be extremely incautious.

With that being said, one might ask himself the following question: are the seller and the buyer at risk of being legally persecuted for a trade that may be deemed as illegal? The answer seems to be no. If data is handled as specified above, there would be no way to associate the seller to its uploaded TEKs at a later time. Data exchanged during the attack would also not directly compromise nor the buyer or the seller[13].

However, there is a problem for a seller who really wants to minimize the chance of getting caught. In fact, since the TEKs proposed by the buyer are posted in clear on the blockchain, authorities may become aware of them and activate ad-hoc procedures monitoring the incriminated TEKs and exploiting the upload authorization process to identify the guilty seller. This, in fact, does not seem to directly contradict the data minimization principle when national security is at stake. If the server getting the TEKs upload monitors the requests (e.g. by storing connection logs) without colluding with the health authority, the seller could be easily incriminated after the TEKs have been detected in the smart contract by just looking at his IP stored together with such request. However, in this case, the usage of an anonymity service like Tor [DMS04] can easily reduce the chance of getting caught. In case that the authorities are colluding, by slightly increasing the complexity of the smart contract, such risk may be completely avoided. It suffices for the buyer to encrypt his TEKs with a public key provided by the seller, who then will use a NIZK proof system[14] to prove that the TEKs encrypted under the specified public key are indeed contained in the list signed with the server's public key. This requires an additional interaction with the buyer, who has to publish the encrypted TEKs. Once again, the seller is protected by a timer which assigns her all the deposits if the buyer does not reply. Efficient Ethereum implementations of NIZK proofs are known in literature, like NIZKs for Σ-protocols [Wil18], Bulletproofs [BAZB20], Zero-Knowledge Range Proofs [KRvW17], zk-SNARKs [ST20a, ZT20b, ZT20a]. An informal description of NIZK proofs is given in Appendix A.3.

Even if the buyer decides to claim the authorship of the attack at a later point in time (e.g., as it usually happens for terrorist attacks) by opening the encrypted values on the blockchain to published TEKs, the seller would not be at risk if data was handled according to the principles of data economy and anonymity. Any evidence based on contact tracing data would be a clear indicator that those principles have been violated. This could result in a big disincentive in using the app, since citizens may think (probably rightfully) that data has been abused or even used for mass surveillance purposes. Finally, we want to point out that even if several researchers raised the concern about the possible birth of black markets [Vau20b] [RGK20], we did not find any document, related to any contact-tracing system, either issued by governments or national security agencies, which deeply evaluates these risks. The Swiss National Cyber Security Center (NCSC) answered concerns expressed by Vaudenay and Vuagnoux [VV20a],

[13]In this analysis, we refer only to contact tracing system data and messages exchanged via the blockchain during the execution of the attack. We do not take into account border-line situations as, for example, the case where there is only a single infected individual. We also ignore additional information that may help investigators figuring out who the seller is, for example how the money are spent after the trade.

[14]In practice, efficient Σ-protocols coupled with Fiat-Shamir heuristic [FS87] can be used, thus getting efficient NIZK proof systems in the random oracle model.

marking them as addressed, without specifying any countermeasure, apart from very generic statements (see [Cen20a] and page 8 [Cen20b]). To the best of our knowledge, no risk analysis ever mentions to monitor the dark web and blockchains looking for suspicious smart contracts. It goes by itself that if blockchains are not monitored, all the extra measures taken in this paragraph to protect the seller are actually not necessary.

3 Exploiting JWTs to Trade TEKs

The Take-TEK attack proposed in Section 2.1 requires that both the buyer and seller deposit some amount of cryptocurrency. It is natural (both for theoretical and practical reasons) to check if there are other effective ways to trade TEKs having one deposit from the buyer only. Indeed, in Take-TEK the seller's coins are blocked until an updated TEKs list is released and this, depending on the particular system, can take from few hours up to one day. Although the seller is guaranteed her money will not get lost, this could in some case be an entry barrier for users who either can not wait so long or own a very little amount of currency. An attack which does not require a seller's deposit would be more appealing for users who do not possess currency, since they would have to make a very low initial investment just covering the transaction fees. Obviously the situation is different for the buyer since he is motivated by a malicious (perhaps catastrophic) intent and he is interested in spending that money anyway. With this regard, note that the presence of a digitally signed TEKs list acts as a proof for the seller of having acted according to the bargain and this unlocks automated trade of uploads. Similarly, the use of a digitally signed token for upload authorization allows the seller to prove possession of a credential that allows to upload TEKs. It follows that proving the possession of a valid token can functionally replace the seller's deposit. Observe that such tokens are not easy to obtain and could limit DoS attacks even more effectively than a monetary deposit. The key idea is that, similarly to what is done for the export files, the smart contract is instantiated with the public key used to verify the JWT. The seller could post her token whose validity is verified by the smart contract (or by the buyer), obtaining the right to redeem the reward afterwards. However, there are some subtleties in this approach:

1. The key for the tokens' signature verification is usually not publicly distributed. Although, for ECDSA, it can be extracted from signed messages as shown in Section 2.2, it is also required to obtain signed message and this is not for free since tokens are given only to infected individuals. However, a buyer might consist of a group of terrorists and it is not unlikely at all that one of them gets infected at some point and that would be the beginning of the attack. Note that just one message-signature pair suffices to extract two possible public keys such that one of them is with very high probability the correct one.
2. It is not clear whether the key used to sign the token may change over time as it happens for the export files. If done frequently, this could reduce the applicability of the attack.
3. The token cannot be posted in clear on the blockchain, otherwise anyone could steal it and use it instead of the seller. However, a simple solution is to omit the digital signature part of the token, proving in zero knowledge the ownership of the digital signature itself. More details on how to do this for ECDSA can be found in Appendix B.2.

3.1 Take-TEK-JWT and Auction-TEK: Attacks Leveraging JWTs

Exploiting long-lived JWT tokens, we propose two smart contracts removing the need for a seller's deposit:

- Take-TEK-JWT: The most straightforward one involves directly replacing the seller's deposit with a valid JWT in the Take-TEK attack. To be more specific, the seller would post

a JWT with the digital signature wiped out, along with a NIZK proof of holding a valid signature, in order to reserve a slot. The smart contract would then verify both the proof and that the JWT is not expired. From this point on, the attack goes on as explained in Section 2.1.

- Auction-TEK: This other variant could give the seller the opportunity to increase her profits even more. Since the token acts as a guarantee of the seller's ability to upload TEKs, the seller may start an auction where different buyers place bids by making deposits. Similarly to what we have seen in Take-TEK, the seller is rewarded if and only if she proves of having uploaded the TEKs specified by the winning buyer. Since the token is valid till after the end of the auction, buyers can be pretty confident the seller is not a fraud. Nevertheless, if the seller does not perform the upload of the specified TEKs, the winning buyer will not lose his money. We remark that buyers can trust the auction only if they know the public key used for verifying the validity of a JWT.

We provide a more detailed description of the above attacks in the following paragraphs.

Playground. We refer to the identity of some player $\mathcal{I}$ as $\mathsf{pk}_{\mathcal{I}}$, which stands also for $\mathcal{I}$'s wallet address. We assume the existence of two signatures schemes such as $(\mathtt{Gen}_{\mathsf{JWT}}, \mathtt{Sign}_{\mathsf{JWT}}, \mathtt{Ver}_{\mathsf{JWT}})$, indicating the scheme used to sign the JWT (we refer to the verification key being used as $\mathsf{vk}_{\mathsf{JWT}}$), and $(\mathtt{Gen}_{\mathbf{T}}, \mathtt{Sign}_{\mathbf{T}}, \mathtt{Ver}_{\mathbf{T}})$ as the scheme used to sign the list of infected TEKs (we refer to the verification being used as $\mathsf{vk}_{\mathbf{T}}$). We also assume the existence of a NIZK proof system $(\mathtt{NIZKProve}, \mathtt{NIZKVer})$ for proving the knowledge of a signature of the scheme being used for the JWT. In particular $\mathtt{NIZKProve}$ takes as an input a pair (s, vk) as a statement, and the value σ as a witness, where s is a JWT string, vk the signature scheme verification key, σ the corresponding signature, and outputs a proof π. The algorithm $\mathtt{NIZKVer}$ takes as an input a statement (s, vk) and a proof π, and outputs 1 if and only if $\mathtt{Ver}(\sigma; \mathsf{vk}) = 1$. Recall that $\mathtt{NIZKVer}$ can verify this last statement without taking σ as an input, but a proof π. The zero-knowledge property of the proof system ensures that π does not leak even a single bit of information related to the signature σ (being a proof of knowledge also guarantees that who generates the proof π actually knows the signature), but the statement can still be successfully verified by $\mathtt{NIZKVer}$ knowing such proof. In both the following attacks, we call k the maximum number of seconds needed for a list of signed TEKs to be published after that $\mathcal{P}$ performs the upload interacting with KS (i.e., providing the buyer's TEKs together with the JWT). Furthermore, we define two auxiliary functions that are useful for both the attacks:

$\mathtt{JWTValidate}(s, \pi_s, \mathsf{vk}_{\mathsf{JWT}}, t_{end})$**:** takes as input a JWT string s together with a proof π_s that $\mathcal{P}$ owns the signature σ_s of s under the key $\mathsf{vk}_{\mathsf{JWT}}$. The algorithm does the following:

- Checks if s is a valid base64 string.
- Parses s into JWT fields, and checks that the value[15] expiration `exp` is less then t_{end} and checks that all the expected claims are correctly set (e.g., in SwissCovid, checks that scope `scope` is set to `currentDayExposed`).
- Check that $\mathtt{NIZKVer}((s, \mathsf{vk}_{\mathsf{JWT}}), \pi_s) = 1$,

If all the checks pass the function returns 1.

$\mathtt{TEKValidate}(\mathbf{T}, s)$**:** Takes as input a TEKs list $\mathbf{T} = (\mathsf{tek}_i, \mathsf{date}_i)_{i \in [n]}$ and a JWT string (omitting the signature) and checks that:

- For each $i \in [n]$, tek_i is in the correct format (i.e., 16 bytes base64 string).
- The JWT s entitles for the upload of the whole $\mathbf{T}$:

[15]The list of JWT claims can be found at `http://jwt.io/introduction`.

- Checks that $\mathbf{T}$ does not contain too much elements both in total and for each given date_i, with respect to the rules of the contact tracing system.
- Checks that, for each $i \in [n]$, date_i is compatible is an allowed date for s. For example, in SwissCovid, the claim `delayedKeyDate` indicates the date for the TEK to be published the following day.

Auction-TEK attack. We do not provide an explicit full smart contract description for this attack, but we assume the functions `Lock` and `SendTeks` to be definable within a smart contract. `Lock` can be triggered by any potential buyer, while `SendTeks` exclusively by the seller. Let $\mathsf{pk}_{\mathcal{P}}$ be the identity of the seller and $\mathsf{pk}_{\mathcal{B}_i}$ the identity of the i-th potential buyer joining the auction. Let t be the smart contract creation time. We define the following functions:

`Constructor`$(\mathsf{vk}_{\mathbf{T}}, \mathsf{vk}_{\mathsf{JWT}}, k, t_1, s, \pi_s)$**:** Triggered by the seller, it takes as input the verification keys $\mathsf{vk}_{\mathbf{T}}$ and $\mathsf{vk}_{\mathsf{JWT}}$ for verifying the signature of $\mathbf{T}$ and of the JWT respectively, a value k indicating the time offset by which the seller must provide the signed TEKs, t_1 a time value specifying when the auction ends, a JWT string s (omitting the digital signature), and a proof $\pi_s \leftarrow \mathtt{NIZKProve}((s, \mathsf{vk}_{\mathsf{JWT}}), \sigma_s)$.

`Lock`$(\mathbf{T}_{\mathcal{B}_i})$**:** It can be triggered by a potential new buyer, and takes as input the buyer's TEKs list $\mathbf{T}_{\mathcal{B}_i} = (\mathsf{tek}_j^{\mathcal{B}_i}, \mathsf{date}_j^{\mathcal{B}_i})_{j\in[n]}$ together with a quantity of coins d_i as a bid for the auction. It runs `TEKValidate`$(\mathbf{T}_{\mathcal{B}_i}, s)$ and, if it outputs 1, compares the bid with the current highest one, and stores the tuple $(i, \mathsf{pk}_{\mathcal{B}_i}, d_i, \mathbf{T}_{\mathcal{B}_i})$ if and only if $d_i > d_{i-1}$. If the checks are successful, the current highest bid is updated and deposit d_{i-1} is returned to $\mathcal{B}_{i-1}$. When time t_1 has passed, `Lock` does not accept any further invocation.

`SendTeks`$(\mathbf{T}, \sigma_{\mathbf{T}})$**:** It can be triggered only by the seller $\mathcal{P}$ before time $t + k$. It takes as input a TEKs list $\mathbf{T} = (\mathsf{tek}_i^{\mathsf{KS}}, \mathsf{date}_i^{\mathsf{KS}})_{i\in[N]}$ where N is the number of published TEKs, together with a signature $\sigma_{\mathbf{T}}$. The algorithm does the following:

- Retrieve a timestamp t' indicating when `SendTeks` is triggered, and check if $t' \leq t+k$.
- Checks that $\mathbf{T}_{\mathcal{B}_j} \subseteq \mathbf{T}$
- Checks that $\mathtt{Ver}_{\mathbf{T}}(\mathbf{T}, \sigma_{\mathbf{T}}; \mathsf{vk}_{\mathbf{T}}) = 1$

If all the checks above are successful, it transfers the d_j coins from the elected buyer's deposit to $\mathcal{P}$'s wallet. It does not accept any further invocation.

For the sake of simplicity, we describe a trivial unfair highest-bid auction, but of course protocols offering more appealing properties can be used [CCL18, GY18]. A description of the attack can be found in Figure 3.

Take-TEK-JWT attack. We address now the scenario in which $\mathcal{B}$ creates the smart contract, and inputs the list of his TEKs $\mathbf{T}_{\mathcal{B}} = (\mathsf{tek}_i^{\mathcal{B}}, \mathsf{date}_i^{\mathcal{B}})_{i\in[n]}$ when triggering the constructor. In this setting, both the functions `Lock` and `SendTeks` are invoked by the seller. In particular:

`Constructor`$(\mathsf{vk}_{\mathbf{T}}, \mathsf{vk}_{\mathsf{JWT}}, p, k, \mathbf{T}_{\mathcal{B}})$**:** The constructor takes as input and stores the verification keys $\mathsf{vk}_{\mathbf{T}}$, $\mathsf{vk}_{\mathsf{JWT}}$, the seller's prize p, a value k indicating the time offset by which the seller must provide the signed TEKs, and the buyer's TEKs list $\mathbf{T}_{\mathcal{B}}$ to be uploaded by the seller.

`Lock`(s, π_s)**:** It can be triggered by a potential seller and takes as input a JWT string s (omitting the digital signature) and a proof of knowledge of a signature of s that verifies under $\mathsf{vk}_{\mathsf{JWT}}$. The auxiliary functions `JWTValidate`$(s, \pi_s, t_{now} + t_{up})$, where t_{now} is the current time and t_{up} is an upper bound on the time needed to complete the upload procedure, and `TEKValidate`$(\mathbf{T}_{\mathcal{B}}, s)$ are invoked. If they both output 1, the address $\mathsf{pk}_{\mathcal{P}}$ is designated as

Auction-TEK Attack

Let us consider a seller $\mathcal{P}$ with wallet $\mathsf{pk}_{\mathcal{P}}$. We indicate with $\mathsf{pk}_{\mathcal{B}_i}$ the identity of the i-th buyer joining the auction. Let $(\mathtt{Gen}_{\mathbf{T}}, \mathtt{Sign}_{\mathbf{T}}, \mathtt{Ver}_{\mathbf{T}})$ be the signature scheme used by the key server KS to sign the TEKs list $\mathbf{T}$, and $\mathsf{vk}_{\mathbf{T}}$ the corresponding verification key. Let $(\mathtt{Gen}_{\mathsf{JWT}}, \mathtt{Sign}_{\mathsf{JWT}}, \mathtt{Ver}_{\mathsf{JWT}})$ be the signature scheme used to issue the JWTs, and $\mathsf{vk}_{\mathsf{JWT}}$ the corresponding verification key. We also assume the existence of a NIZK proof system $(\mathtt{NIZKProve}, \mathtt{NIZKVer})$ for proving the knowledge of a signature under the signature scheme $(\mathtt{Gen}_{\mathsf{JWT}}, \mathtt{Sign}_{\mathsf{JWT}}, \mathtt{Ver}_{\mathsf{JWT}})$.
The protocol of the attack works as follows:

1. $\mathcal{P}$ creates a smart contract featuring the functions `Lock` to be triggered by any potential buyer and `SendTeks` to be triggered only by $\mathcal{P}$ herself. When triggering the constructor of the smart contract, $\mathcal{P}$ sends the values $(\mathsf{vk}_{\mathbf{T}}, \mathsf{vk}_{\mathsf{JWT}}, k, t_1, s, \pi_s)$ to it. It automatically instantiates a value t with the smart contract creation time.
2. $\mathcal{B}_i$ checks that $\mathsf{vk}_{\mathsf{JWT}}$ and $\mathsf{vk}_{\mathbf{T}}$ are the correct verification keys and invokes $\mathtt{JWTValidate}(s, \mathsf{vk}_{\mathsf{JWT}}, \pi_s, t_1 + t_{up})$ to check if the JWT is valid, where the value t_{up} is an upper bound on the time needed to complete an upload. If `JWTValidate` returns 1, $\mathcal{B}_i$ triggers $\mathtt{Lock}(\mathbf{T}_{\mathcal{B}_i})$ by sending d_i coins as a payment, thus placing his bid.
3. Once time t_1 elapses, $\mathtt{Lock}(\mathbf{T}_{\mathcal{B}_i})$ does not accept any further invocation.
4. If $\mathcal{P}$ triggers $\mathtt{SendTeks}(\mathbf{T}, \sigma_{\mathbf{T}})$ before time $t + k$ and all the conditions are met (i.e., $\mathbf{T}_{\mathcal{B}} \subseteq \mathbf{T}$ and $\mathtt{Ver}_{\mathsf{KS}}(\mathbf{T}, \sigma_{\mathbf{T}}; \mathsf{vk}_{\mathbf{T}}) = 1$), the smart contract automatically transfers d_i coins to $\mathcal{P}$'s wallet.
5. If $\mathcal{P}$ does not provide $(\mathbf{T}, \sigma_{\mathbf{T}})$ before time $t + k$, d_i coins are returned back to $\mathcal{B}_i$.

Figure 3: Auction-TEK Attack: exploiting long-lived JWTs to sell uploads via auctions.

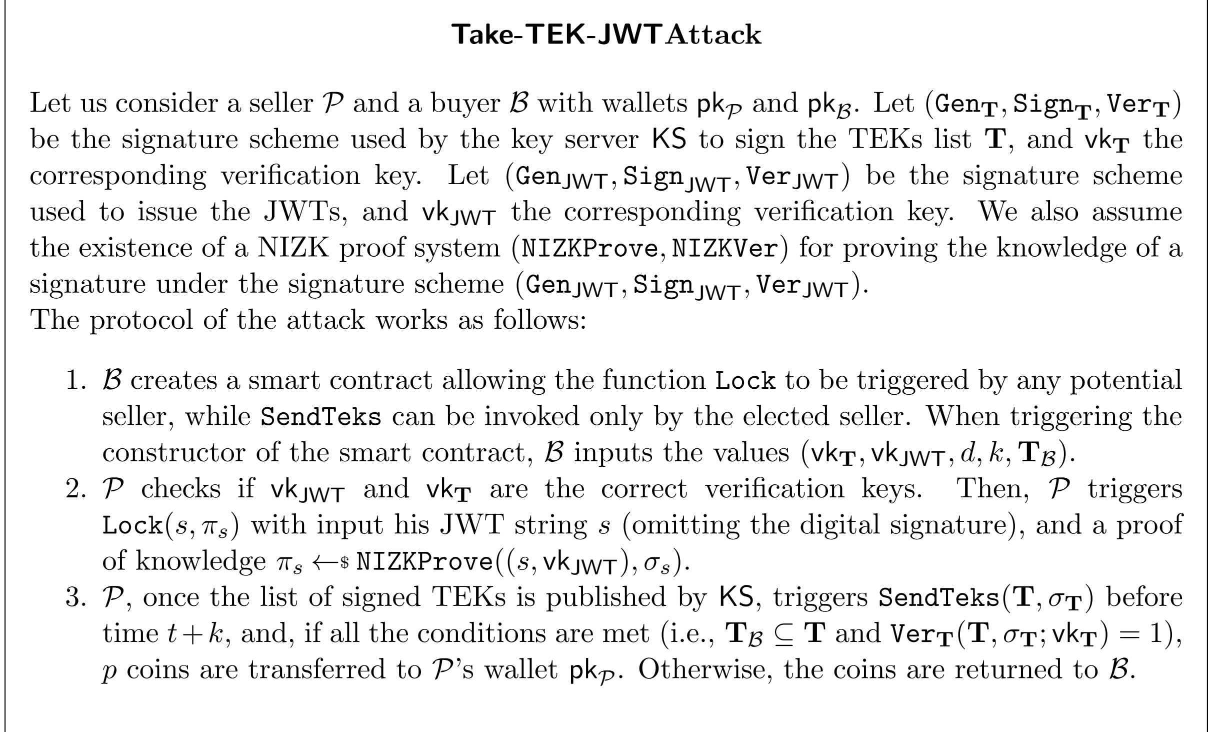
the elected seller's address. The pair (s, π_s) is stored in the smart contract and t is set as the function invocation time. It does not accept any further invocation.

$\mathtt{SendTeks}(\mathbf{T}, \sigma_{\mathbf{T}})$**:** It can be triggered only by the elected seller $\mathcal{P}$ before time $t+k$ and takes as input a list of TEKs $\mathbf{T}$ together with a signature $\sigma_{\mathbf{T}}$. The algorithm does the following:

- Retrieve a timestamp t' indicating when `SendTeks` is triggered, and check if $t' \leq t+k$.
- Check that $\mathbf{T}_{\mathcal{B}} \subseteq \mathbf{T}$
- Check that $\mathtt{Ver}_{\mathbf{T}}(\mathbf{T}, \sigma_{\mathbf{T}}; \mathsf{vk}_{\mathbf{T}}) = 1$

If all the above checks are successful, it transfers d coins to the elected $\mathcal{P}$'s wallet $\mathsf{pk}_{\mathcal{P}}$. It does not accept any further invocation.

We provide a detailed description of the attack in Figure 4.



Take-TEK-JWT Attack

Let us consider a seller $\mathcal{P}$ and a buyer $\mathcal{B}$ with wallets $\mathsf{pk}_{\mathcal{P}}$ and $\mathsf{pk}_{\mathcal{B}}$. Let $(\mathtt{Gen}_{\mathbf{T}}, \mathtt{Sign}_{\mathbf{T}}, \mathtt{Ver}_{\mathbf{T}})$ be the signature scheme used by the key server KS to sign the TEKs list $\mathbf{T}$, and $\mathsf{vk}_{\mathbf{T}}$ the corresponding verification key. Let $(\mathtt{Gen}_{\mathsf{JWT}}, \mathtt{Sign}_{\mathsf{JWT}}, \mathtt{Ver}_{\mathsf{JWT}})$ be the signature scheme used to issue the JWTs, and $\mathsf{vk}_{\mathsf{JWT}}$ the corresponding verification key. We also assume the existence of a NIZK proof system $(\mathtt{NIZKProve}, \mathtt{NIZKVer})$ for proving the knowledge of a signature under the signature scheme $(\mathtt{Gen}_{\mathsf{JWT}}, \mathtt{Sign}_{\mathsf{JWT}}, \mathtt{Ver}_{\mathsf{JWT}})$.
The protocol of the attack works as follows:

1. $\mathcal{B}$ creates a smart contract allowing the function `Lock` to be triggered by any potential seller, while `SendTeks` can be invoked only by the elected seller. When triggering the constructor of the smart contract, $\mathcal{B}$ inputs the values $(\mathsf{vk}_{\mathbf{T}}, \mathsf{vk}_{\mathsf{JWT}}, d, k, \mathbf{T}_{\mathcal{B}})$.
2. $\mathcal{P}$ checks if $\mathsf{vk}_{\mathsf{JWT}}$ and $\mathsf{vk}_{\mathbf{T}}$ are the correct verification keys. Then, $\mathcal{P}$ triggers $\mathtt{Lock}(s, \pi_s)$ with input his JWT string s (omitting the digital signature), and a proof of knowledge $\pi_s \leftarrow\$ \mathtt{NIZKProve}((s, \mathsf{vk}_{\mathsf{JWT}}), \sigma_s)$.
3. $\mathcal{P}$, once the list of signed TEKs is published by KS, triggers $\mathtt{SendTeks}(\mathbf{T}, \sigma_{\mathbf{T}})$ before time $t+k$, and, if all the conditions are met (i.e., $\mathbf{T}_{\mathcal{B}} \subseteq \mathbf{T}$ and $\mathtt{Ver}_{\mathbf{T}}(\mathbf{T}, \sigma_{\mathbf{T}}; \mathsf{vk}_{\mathbf{T}}) = 1$), p coins are transferred to $\mathcal{P}$'s wallet $\mathsf{pk}_{\mathcal{P}}$. Otherwise, the coins are returned to $\mathcal{B}$.

Figure 4: Take-TEK-JWT attack: one deposit only, exploiting long-lived JWTs.

3.2 Worry-Free Seller with JWT

As in Section 2.2, one might wonder whether publishing (part of) the JWT poses additional threats for the seller in terms of chances of being legally persecuted. Obviously, for both Take-TEK-JWT and Auction-TEK attacks, the risks related to publishing the TEKs in clear can be addressed as shown in Section 2.2. On the other hand, encrypting the JWT and proving statements about it in zero-knowledge would not be as practical as it is for the TEKs; this is for two reasons:

- Proving the knowledge of an ECDSA signature when both the message and the simulatable part of the signature (i.e., r) are also witnesses is a way more involved process. To this regard, Backes et al. [BHH+19] provide a construction to prove such statements non-interactively and without trusted setups. (Nevertheless, it may be not practical enough to be used on-chain, both in terms of proof size and verification time.)

- Proving knowledge of a valid signature is not enough because the signed message must also comply to certain requirements. In particular, statements about JWT claims should be proved (e.g., the JWT must not be expired). This would require quite a lot of engineering and the use of custom techniques.

Since TEKs are uploaded only after the JWT is revealed, ad-hoc monitoring procedures could be put in place for the JWT as well. However, according to data minimization principles, the system should not build mappings between authorization codes, JWTs, and identifiable information; especially when there is no reason to believe the user is dishonest. Google also recommends to not associate identifiable information with this data [Goo20d]. This implies that authorities should not possess any mapping between a JWT and the identity of an infected user by the time the JWT gets published.

Comparing the Take-TEK attack with Auction-TEK and Take-TEK-JWT attacks, the first one offers better privacy guarantees to the seller. In fact, in the Take-TEK attack, if the buyer does not open the encrypted TEKs, the seller is protected also against malicious servers retaining more information than advertised. Obviously, if blockchains are not monitored there are no risks at all for the seller, regardless of the attack. Finally, we want to point out that all the attacks proposed in the previous sections are practical and guarantee honest sellers to be eventually rewarded. If the seller is more concerned of getting caught, she could look for a smart contract offering a Take-TEK attack, while if she wants to avoid depositing coins she could go for JWT attacks.

3.3 SwissCovid's Additional Risk: Next-Day JWT

As explained in Section 2 GAEN API's goal is to provide only an efficient decentralized pseudonyms management through BLE advertisements allowing to check for risk exposure and to export TEKs. A digital contact tracing system using GAEN has still some tasks to accomplish and one of them is the upload authorization protocol. In this section we illustrate the choice of SwissCovid that consists of digitally signed authorization tokens. We will show that this choice, can also be abused to realize terrorist attacks. The protocol involves four entities: the infected user U (and his device), the health authority HA, the verification server VS, and the key server KS. The protocol proceeds as follows.

1. Once U is diagnosed the disease, HA asks VS for a numerical code, called covidcode, and forwards it back to U (e.g., via a phone call or a text message).
2. U inputs the covidcode to his device which hands it to VS in exchange for a token that can be used to upload the TEKs to KS. This token is implemented via a JSON Web Token (JWT) [16] digitally signed by VS itself. Among other claims, these tokens contain a validity period.
3. When U decides to share his TEKs, he submits the JWT along with his TEKs to KS. Since GAEN does not allow to extract the TEK of the current day[17], a JWT valid for uploading an additional TEK the next day is also issued by KS and sent to U.
4. KS verifies the correctness of the data and eventually publishes the TEKs distributing the appropriate export files.

[16] A JSON Web Token (JWT), is a compact way of representing and transferring claims. The claims in a JWT are encoded as a JSON object that can be digitally signed or authenticated with a MAC. A more detailed description can be found at `https://tools.ietf.org/html/rfc7519`.

[17] The TEK of the current day could be used by malicious entities to generate valid pseudonyms linked to a positive test, thus causing fake at-risk notifications. However, this limitation might change since the current version of GAEN (v1.6) allows to extract it in debug mode. Google's documentation state this feature will be soon activated in production (see "Release Notes" in [Goo20e]).

The key point here is that a numerical non-publicly verifiable code is exchanged for a digitally signed token. Therefore, if a token has a long enough validity, it can be used effectively in smart contracts to prove of having the capability to upload TEKs. The JWT issued by VS has a validity period of 5 minutes and uses an RSA signature (see [FOoITF20]), while the one issued by KS is valid for two days (see row 207 in [ST20c]) and uses ECDSA as signature algorithm. This last JWT easily accommodates the attacks discussed in Section 3, allowing to trade a TEK with the first upload's day as date. We also point out that the use of signed tokens seems to be a practice used in other contact tracing systems (e.g., Corona-Warn [CT20c]), the attacks may apply also to them if JWTs are valid for a long-enough time interval. However, we want to stress that we have focused our research on Immuni and SwissCovid only. Immuni uses only a numerical non-publicly verifiable code, and thus is not affected by this additional issue.

4 No Signatures? Connecting Smart Contracts to TLS Sessions

All the attacks we have shown up to now rely on the fact that a digital signature is used to authorize uploads. Additionally, the ability to extract the public key from signed messages also plays a key role. Therefore, one might think that to protect GAEN systems the public key should remain hidden and the signature scheme should be such that extracting the public key from message-signature pairs is hard. In this way, due to the inability of allowing a smart contract to verify that a TEK is officially in a list of infected TEKs, all the attacks proposed up to now would fail. However, things are not so easy and designers of digital contact tracing systems should be careful. The previous smart contracts were exploiting the public verifiability of the signatures because this is what is used in GAEN systems. If a different method is used, it might be abused again.

Indeed, just to present one more instantiation of a terrorist attack, we show that TLS oracles can be used to prove to a smart contract that an upload was successfully performed, therefore without relying on signatures of TEKs.

4.1 Decentralized Oracles

Recently, Zhang et al. [ZMM+19], introduced the concept of Decentralized Oracles. Roughly, an oracle is an entity that can be queried by a client to interact with a TLS server and help the client proving statements about the connection transcript. Previously known oracle constructions rely on trusted/semi-trusted Execution Environments [ZCC+16], thus not giving any help in our case. DECO [ZMM+19] is the first work where a fully-decentralized construction is proposed for specific ciphersuites as CBC-HMAC and AES-GCM coupled with DH key exchange with ephemeral secrets. We recall that a TLS connection is divided in two parts: a handshake phase where key exchange is performed, and a phase during which encrypted messages are exchanged, data is encrypted/decrypted by the client/server using the key exchanged in the previous phase. GAEN servers usually accept Elliptic-Curve Diffie-Hellman key Exchange (ECDHE) for the first phase, while for the second phase some servers accept only AES-GCM (e.g., Immuni), whereas others, like SwissCovid's one, accept also CBC-HMAC as a ciphersuite. To guarantee integrity of data, the plaintext is usually compressed and a MAC on the compressed string is calculated (using once again a key derived from the DH exchanged key).

Decentralized Key-Exchange. We provide below an informal description on how key-exchange is executed in DECO for ECDHE. This phase is called Three Party Handshake (3PHS).

We assume three entities: a prover $\mathcal{P}$, a verifier $\mathcal{V}$ and a server S. $\mathcal{P}$ and $\mathcal{V}$ jointly act as a TLS client. The overall idea of DECO is that the prover and verifier, after performing some

two-party computations, compute shares of the exchanged key, while the server computes the entire key without even noticing that $\mathcal{P}$ and $\mathcal{V}$ are two distinct interacting entities.

When using CBC-HMAC, the keys $k^{\mathtt{MAC}}_{\mathcal{P}}$, $k^{\mathtt{MAC}}_{\mathcal{V}}$ (such that $k^{\mathtt{MAC}}_{\mathcal{P}} + k^{\mathtt{MAC}}_{\mathcal{V}} = k^{\mathtt{MAC}}$) are learned by $\mathcal{P}$ and $\mathcal{V}$ respectively, while $k^{\mathtt{Enc}}$ is only known to $\mathcal{P}$. When using AES-GCM, the same key is used for both encryption and MAC, therefore both $\mathcal{P}$ and $\mathcal{V}$ get just a share of it. While $\mathcal{P}$ and $\mathcal{V}$ only learn their secret shares of the key, the server S gets to know both $k^{\mathtt{Enc}}$ and $k^{\mathtt{MAC}}$. From its perspective, the server is communicating only with a single entity knowing the same exchanged key. The key exchange phase works as follows:
Let G be an EC group generator.

- $\mathcal{P}$ generates a random nonce r_c and sends it to S in the ($\mathsf{ClientHello}$) message.
- When receiving a certificate, the server nonce r_s, and a signed DH share $Y_\mathsf{S} = s_\mathsf{S} \cdot G$ from S, $\mathcal{P}$ checks the certificate and the signature, and forwards them to $\mathcal{V}$.
- $\mathcal{V}$ samples a DH secret $s_\mathcal{V}$ and sends his DH share $Y_\mathcal{V} = s_\mathcal{V} \cdot G$ to $\mathcal{P}$.
- $\mathcal{P}$ samples her DH secret $s_\mathcal{P}$, calculates her DH share $Y_\mathcal{P} = s_\mathcal{P} \cdot G$, calculates the combined DH share $Y = Y_\mathcal{P} + Y_\mathcal{V}$, and sends Y to S.

Finally, S computes the DH exchanged key as $Z = s_\mathsf{S} \cdot Y$. $\mathcal{P}$ and $\mathcal{V}$ will compute their secret shares of Z as $Z_\mathcal{P} = s_\mathcal{P} \cdot Y_\mathsf{S}$ and $Z_\mathcal{V} = s_\mathcal{V} \cdot Y_\mathsf{S}$. Note that $Z_\mathcal{P} + Z_\mathsf{S} = Z$, where $+$ is the EC group operation. Now that $\mathcal{P}$ and $\mathcal{V}$ have secret shares of EC points, they use 2PC to evaluate a PRF (that we call TLS-PRF) to derive the keys. The authors face and solve several challenges in order to derive keys efficiently via 2PC. We do not cover this part, a more detailed description can be found in [ZMM+19].

Encrypted communication. At the end of the 3PHS, $\mathcal{P}$ and $\mathcal{V}$ have to engage in a 2PC protocol to correctly calculate the MAC and the encryption on the plaintext to be sent to the server, without revealing the shares to each other. Privacy of the plaintext is also ensured with respect to $\mathcal{V}$. For CBC-HMAC, the encryption is computed exclusively by $\mathcal{P}$ who holds the encryption key. The authors provide hand-optimized protocols which are much more efficient then the ones obtained by directly applying 2PC techniques. Since $\mathcal{P}$ and $\mathcal{V}$ must cooperate also for encryption, the 2PC protocol for AES-GCM is a lot slower than the one for CBC-HMAC.

Proving Statements. An important feature of DECO is that $\mathcal{P}$, when the communication with S ends, can prove, in zero knowledge, statements on the communication transcript in a clever and efficient way. However, to make their protocol practical for our goal, we give up on maintaining the transcript private. As a result, we do not cover this part of DECO which can be found in [ZMM+19]. In Section 4.2, we provide a description on how to adapt, by simplifying it, DECO to our scenario.

4.2 A Smart Contract Oracle

The starting idea of the attack we introduce in this section is to make the smart contract play the role of the DECO verifier. In this way, the smart contract would be able to verify that the intended communication between the seller and the server took place and to reward the seller accordingly. Unfortunately, doing so directly by plugging DECO into a smart contract is not possible for several reasons. For example, DECO requires a lot of interaction and to run intensive 2PC related tasks, to sample random values and most notably to maintain a private state. Therefore, we run the DECO protocol off-chain with the seller acting as prover and the buyer as verifier, but we guarantee no party is able to cheat (i.e., the seller is paid if and only if she performs the upload of the requested TEKs) by binding its execution to

the state of the smart contract itself. Furthermore, we guarantee the privacy of the messages exchanged between the server and the prover only until their TLS connection is open. After the communication ends, the seller proves that she acted honestly by providing the application-level messages exchanged with the server, along with the corresponding MAC tags w.r.t. the MAC key which is bound to the smart contract. To be more specific, the smart contract freezes a share of the MAC key and the seller has to show a communication transcript (i.e., the messages exchanged with the server and corresponding MAC tags) which is consistent with such share. Privacy of the upload request message to be sent to the server is crucial while the TLS session is open because the verifier may abort the protocol and use the authorization token of the prover to upload data by himself without paying out the promised reward. On the other hand, making the communication public after it took place does not endanger the prover, apart from the considerations made in Sections 2.2 and 3.2, and makes the verification procedure much more practical. What we need is that the shares of the prover and the verifier are kept private until the end of the protocol, and then revealed to the smart contract, along with other information, for verification and reward paying. In addition, the TLS session timeout should be big enough to allow for the 2PC execution. To this regard, Zhang et. al already verified the practical feasibility of their protocol [ZMM+19]. Obviously, $\mathcal{P}$ must know how to reach $\mathcal{V}$ to carry out the protocol. To address concerns regarding anonymity, $\mathcal{V}$ may set up a TOR hidden service[18]. Using hidden services may significantly slow down the process, however we found both Immuni and SwissCovid servers to give a generous time out window of two hours[19] (see Appendix D).

From now on, we refer to the seller and the buyer also as prover $\mathcal{P}$ and verifier $\mathcal{V}$ respectively; we denote the server as S. In the following, we explain the attack in more detail for the CBC-HMAC ciphersuite. At smart contract creation time, $\mathcal{V}$ posts the DH share $Y_{\mathcal{V}} = s_{\mathcal{V}} \cdot G$ he is willing to use during the 3PHS, along with requested TEKs (and dates).

First, $\mathcal{P}$ transacts on the smart contract to reserve a time slot of duration t_1 by which a DECO protocol run must be performed together with $\mathcal{V}$ and S, and the data needed to redeem the reward must be posted on the smart contract by $\mathcal{P}$. If time t_1 elapses, $\mathcal{P}$ loses her slot. This reservation mechanism is needed to prevent $\mathcal{V}$ from getting back the reward while an honest $\mathcal{P}$ performs the upload of the requested TEKs. In fact, the verifier could also act as a prover and simulate a reward-paying interaction with the server to the smart contract, which would have no mean to distinguish it from a fake one. By adding a reservation mechanism, we are sure a malicious $\mathcal{V}$ cannot play a simulated transcript in the smart contract while honest $\mathcal{P}$ is performing with him the DECO protocol run. Furthermore, since the communication for the upload between server and the prover consists of just a single query followed by a single response, it is not possible for a cheating verifier to make the timer expire avoiding to pay the prover while at the same time the upload of the TEKs happens successfully. In fact, once the $\mathcal{V}$ cooperates with $\mathcal{P}$ to build a valid request, S will reply to $\mathcal{P}$ independently of what $\mathcal{V}$ does, thus giving $\mathcal{V}$ all she needs to redeem the reward.

When executing the 3PHS, $\mathcal{P}$ checks that the value $Y'_{\mathcal{V}}$ sent by $\mathcal{V}$ during the handshake corresponds to the value $Y_{\mathcal{V}}$ posted to the smart contract. This prevents $\mathcal{V}$ from providing an erroneous DH share and blaming $\mathcal{P}$ for it. If this is not the case, $\mathcal{P}$ aborts. Since no upload message has been sent to the server yet, no party gains advantage from this operation. If $\mathcal{V}$'s share is correct (i.e., $Y_{\mathcal{V}} = Y'_{\mathcal{V}}$), parties engage in the communication with S and jointly compute the MAC (via 2PC as in [ZMM+19]) on the upload request m_c generated by $\mathcal{P}$. If the connection ends successfully[20], the elected $\mathcal{P}$ posts (only who reserved this slot is allowed

[18]More on TOR hidden services can be found at `https://2019.www.torproject.org/docs/onion-services`.

[19]Interestingly, in June the timeout of a TLS session with both Immuni and SwissCovid upload servers was limited to 5 minutes, but it has been then extended to two hours.

[20]This can be inferred from the communication transcript. For example, as in SwissCovid [ST20c], S may reply

to post this message) to the smart contract the following:

- The entire communication transcript, that is (m_c, m_s) together with the MACs (θ_c, θ_s), calculated by the client(s) $\mathcal{P} \leftrightarrow \mathcal{V}$ and the server S respectively.
- The prover's secret $s_{\mathcal{P}}$.
- The DH share of the server Y_{S} received during the 3PHS.

Then, the smart contract starts a timer t_2 indicating the maximum time $\mathcal{V}$ has to reveal his secret $s_{\mathcal{V}}$. In the case he does not do that, the prize is automatically transferred to the seller. Now $\mathcal{V}$, in order to avoid paying the prize for no reason, reveals $s_{\mathcal{V}}$. The smart contract does the following:

- Check that $Y_{\mathcal{V}} = s_{\mathcal{V}} \cdot G$ and if not, transfer the prize to $\mathcal{P}$.
- If the check passes, reconstruct the secret Z from $s_{\mathcal{V}}, s_{\mathcal{P}}, Y_s$, and apply TLS-PRF to derive the MAC key $k^{\mathtt{MAC}}$.

Now the smart contract has everything it needs to check that the fields inside the uploading message m_c (from the prover to the server) are correct (i.e., the buyer's TEK are present) and the response message (from the server to the prover) is positive, and that the MACs (θ_c, θ_s) verify w.r.t. $k^{\mathtt{MAC}}$. If all the checks pass, the prize is transferred to $\mathcal{P}$, otherwise $\mathcal{P}$ gains no prize and the deposit is returned back to $\mathcal{V}$.

As mentioned before, $\mathcal{V}$ is not encouraged to provide a different public key w.r.t. the one he used in DECO execution, otherwise $\mathcal{P}$ will just abort. On the other hand, the prover is not able to earn a reward without uploading the promised TEKs. In fact, the probability for the prover to come up with a pair (m'_c, θ'_c) (resp. (m'_s, θ'_s)) that verifies under the key $k'^{\mathtt{MAC}}$ derived from $Z' = Z'_{\mathcal{P}} + Z'_{\mathcal{V}}$ with $Z'_{\mathcal{P}} := s'_{\mathcal{P}} \cdot Y'_{\mathsf{S}}$ and $Z_{\mathcal{V}} := s_{\mathcal{V}} \cdot Y'_{\mathsf{S}}$ is negligible due to the fact that $s_{\mathcal{P}}$ is fixed and honestly generated, thus randomizing Z', hence $k'^{\mathtt{MAC}}$.

CBC-HMAC vs AES-GCM. As mentioned before, differently from CBC-HMAC, AES-GCM relies on the same key for both encryption and MACs. The impact of AES-GCM is twofold: 1) more computation is needed to perform the required 2PC to calculate messages from/to the server, due to the AES algorithm itself, 2) the prover does not learn the encryption key after 3PHS, meaning that both encryption and decryption must be done via 2PC as well. On the smart contract side, this difference boils down to a lack of fairness. After $\mathcal{V}$ and $\mathcal{P}$ have calculated together the upload message and sent it then to S, $\mathcal{V}$ could decide not to help the prover to decrypt the server's response. Now, $\mathcal{P}$ has no witness in her hands to give to the smart contract in order to prove that she has correctly performed the TEKs upload. As a result, she cannot redeem the prize. The problem can be easily solved by giving to the smart contract the burden of decrypting the server's ciphertext. In our approach, $\mathcal{V}$ must commit to his key and open it later. When this happens, the server reconstructs the MAC/encryption key, decrypts the ciphertext, do the necessary checks, and pay the prize to $\mathcal{P}$. The CBC-HMAC version of DECO is way faster then the AES-GCM one. However, looking at practical evaluations made by the authors [ZMM+19] it is reasonable to think that all their solutions may fit in the time window given by contact tracing servers (e.g., 2 hours in Immuni and SwissCovid) for the TLS connection, even when hiding $\mathcal{V}$ through TOR hidden services.

We want to stress that this completely different instantiation of a terrorist attack is only an example of attacks that could potentially be mounted on GAEN system even when the public verifiability of the list of infected TEKs is removed. Therefore what we aim to show here is that designers of digital contact tracing systems should take also these alternative attacks into

$\mathcal{P}$ with either a success message such as "200 OK" or an error message.

account. Ideally, the protocol designers should not focus on specific realizations of an attack, but instead should prove that the protocol is secure against any automated instantiation of a terrorist attack.

A note on DoS attacks. As noted before, it is important to prevent DoS attack run by sellers who actually do not have the right to upload TEKs and end up by just wasting buyer's precious time. In the attack explained in this section, this protection is not provided: a malicious $\mathcal{P}$ could post fake transcripts, forcing $\mathcal{V}$ to open his DH share in order to not lose his deposit. What is worse is that since it has to be opened for verification, $\mathcal{V}$'s share cannot be reused in another DECO protocol run. The above problem can be solved by introducing the two following key modifications:

1. The moment the slot is reserved is moved after the 3PHS of DECO, when $\mathcal{P}$ gets to know the MAC tag θ_c on the upload request m_c to be sent to S later on. $\mathcal{P}$ forwards (m_c, θ_c) to S only after the slot reservation is confirmed on the blockchain.
2. When reserving a slot, $\mathcal{P}$ must attach a certain deposit of currency. Such deposit is returned back to $\mathcal{P}$ if the final verification is successful, otherwise it is transferred to $\mathcal{V}$.

Since the reply from S will be received by $\mathcal{P}$ regardless of what $\mathcal{V}$ does, the first step guarantees that $\mathcal{P}$ will be able to eventually redeem the prize when she makes a reservation. On the other hand, the money deposit effectively discourage any attempt on DoS. Nevertheless, this construction introduces an additional waiting time, accounting for the blockchain verification time, in the overall DECO run. This means that the TLS time out offered by S should be long enough to accommodate also this additional waiting time.

5 Acknowledgments

We thank the first author of DECO [ZMM+19] Fan Zhang for all the clarifications about their paper, Stephen Farrell of the TACT project [LF20] for his help on how to gather contact tracing data, Serge Vaudenay and Martin Vuagnoux for useful information about the implementation and configuration of SwissCovid. This research is supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No 780477 (project PRIViLEDGE).

References

[ABIV20] Gennaro Avitabile, Vincenzo Botta, Vincenzo Iovino, and Ivan Visconti. Towards defeating mass surveillance and sars-cov-2: The pronto-c2 fully decentralized automatic contact tracing system. *IACR Cryptol. ePrint Arch.*, 2020:493, 2020.

[And20] Ross Anderson. Contact tracing in the real world. `https://www.lightbluetouchpaper.org/2020/04/12/contact-tracing-in-the-real-world/`, 2020. Accessed: 2020-09-18.

[App20a] Apple. detectExposures method description from Apple's EN documentation. `https://developer.apple.com/documentation/exposurenotification/enmanager/3586331-detectexposures`, 2020. Accessed: 2020-08-23.

[App20b] Apple. Setting up an Exposure Notification server. `https://developer.apple.com/documentation/exposurenotification/setting_up_an_exposure_notification_server`, 2020. Accessed: 2020-08-23.

[Aus20] The Crypto Group IST Austria. Inverse-sybil attacks in automated contact tracing. *IACR Cryptol. ePrint Arch.*, 2020:670, 2020.

[BAZB20] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In *Financial Cryptography*, pages 423–443, 2020.

[BCC+20] Xavier Bonnetain, Anne Canteaut, Véronique Cortier, Pierrick Gaudry, Lucca Hirschi, Steve Kremer, Stéphanie Lacour, Matthieu Lequesne, Gaëtan Leurent, Léo Perrin, André Schrottenloher, Emmanuel Thomé, Serge Vaudenay, and Christophe Vuillot. Anonymous tracing, a dangerous oxymoron. a risk analysis for non-specialists. https://tracing-risks.com/docs/tracing-risks.pdf, 2020. Accessed: 2020-09-07.

[BDF+20] Lars Baumgärtner, Alexandra Dmitrienko, Bernd Freisleben, Alexander Gruler, Jonas Höchst, Joshua Kühlberg, Mira Mezini, Markus Miettinen, Anel Muhamedagic, Thien Duc Nguyen, Alvar Penning, Dermot Frederik Pustelnik, Filipp Roos, Ahmad-Reza Sadeghi, Michael Schwarz, and Christian Uhl. Mind the GAP: security & privacy risks of contact tracing apps. *CoRR*, abs/2006.05914, 2020.

[BHH+19] Michael Backes, Lucjan Hanzlik, Amir Herzberg, Aniket Kate, and Ivan Pryvalov. Efficient non-interactive zero-knowledge proofs in cross-domains without trusted setup. In Dongdai Lin and Kazue Sako, editors, *PKC 2019, Part I*, volume 11442 of *LNCS*, pages 286–313. Springer, Heidelberg, April 2019.

[CCL18] Yi-Hui Chen, Shih-Hsin Chen, and Iuon-Chang Lin. Blockchain based smart contract for bidding system. In *2018 IEEE International Conference on Applied System Invention (ICASI)*, pages 208–211. IEEE, 2018.

[Cen20a] Swiss National Cyber Security Center. Security issue submission [inr-4434]. detailed analysis. https://www.melani.admin.ch/dam/melani/de/dokumente/2020/INR-4434_NCSC_Risk_assessment.pdf.download.pdf/INR-4434_NCSC_Risk_assessment.pdf, 2020. Accessed: 2020-08-23.

[Cen20b] Swiss National Cyber Security Center. Swisscovid proximity tracing system - public security test. https://www.melani.admin.ch/dam/melani/de/dokumente/2020/SwissCovid_Public_Security_Test_Current_Findings.pdf.download.pdf/SwissCovid_Public_Security_Test_Current_Findings.pdf, 2020. Accessed: 2020-08-23.

[Clu20] Chaos Computer Club. 10 requirements for the evaluation of "contact tracing" apps. https://www.ccc.de/en/updates/2020/contact-tracing-requirements, 2020. Accessed: 2020-08-23.

[Com20] European Commission. Guidance on apps supporting the fight against COVID 19 pandemic in relation to data protection. *Official Journal of the European Union*, 2020.

[CT20a] Corona-Warn's Team. Corona-warn-app solution architecture. https://github.com/corona-warn-app/cwa-documentation/blob/master/solution_architecture.md, 2020. Accessed: 2020-08-23.

[CT20b] Corona-Warn's Team. Criteria for the evaluation of contact tracing apps. https://github.com/corona-warn-app/cwa-documentation/blob/ec703906c109bd7c3cc84bc361b7e703b20650ea/pruefsteine.md, 2020. Accessed: 2020-08-23.

[CT20c] Corona-Warn's Team. Software design verification server. https://github.com/corona-warn-app/cwa-verification-server/blob/f43c6be9b9cb3017dc77c38e50211aaa32fd33c4/docs/architecture-overview.md, 2020. Accessed: 2020-08-23.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In *USENIX*, pages 303–320, 2004.

[DR20a] Paul-Olivier Dehaye and Joel Reardon. Proximity tracing in an ecosystem of surveillance capitalism. *CoRR*, abs/2009.06077, 2020.

[DR20b] Paul-Olivier Dehaye and Joel Reardon. Swisscovid: a critical analysis of risk assessment by swiss authorities. *CoRR*, abs/2006.10719, 2020.

[DT20] DP-3T's Team. Decentralized privacy-preserving proximity tracing. *https://github.com/DP-3T/documents/blob/master/DP3T%20White%20Paper.pdf*, 2020.

[eN20] EU eHealth Network. European proximity tracing. an interoperability architecture. https://ec.europa.eu/health/sites/health/files/ehealth/docs/mobileapps_interoperabilitydetailedelements_en.pdf, 2020. Accessed: 2020-09-17.

[FOoITF20] Systems Federal Office of Information Technology and Telecommunication FOITT. Swisscovid custom token provider. https://github.com/admin-ch/CovidCode-Service/blob/18b8f3f6a268d9d245c69ea61fcc146f73b8e11f/src/main/java/ch/admin/bag/covidcode/authcodegeneration/service/CustomTokenProvider.java, 2020. Accessed: 2020-09-05.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *STOC*, pages 291–304, 1985.

[Goo20a] Google. Exposure Key export file format and verification. https://developers.google.com/android/exposure-notifications/exposure-key-file-format, 2020. Accessed: 2020-08-23.

[Goo20b] Google. Exposure Notification Cryptography Specification. https://blog.google/documents/69/Exposure_Notification_-_Cryptography_Specification_v1.2.1.pdf, 2020. Accessed: 2020-08-23.

[Goo20c] Google. Exposure Notification Reference Key Server. https://google.github.io/exposure-notifications-server/, 2020. Accessed: 2020-08-23.

[Goo20d] Google. Exposure Notification verification server. https://developers.google.com/android/exposure-notifications/verification-system, 2020. Accessed: 2020-08-23.

[Goo20e] Google. Exposure Notifications API. https://developers.google.com/android/exposure-notifications/exposure-notifications-api, 2020. Accessed: 2020-08-23.

[GY18] Hisham S. Galal and Amr M. Youssef. Verifiable sealed-bid auction on the ethereum blockchain. In *Financial Cryptography*, pages 265–278, 2018.

[Har20] Charlie Hartmann. Taking the covid-19 test in switzerland. https://livingin.swiss/2020/09/11/taking-the-covid-19-test-in-switzerland/, 2020. Accessed: 2020-09-17.

[IT20] Immuni's Team. Immuni's high-level description. https://github.com/immuni-app/immuni-documentation, 2020. Accessed: 2020-08-23.

[KNW20] Majid Khabbazian, Tejaswi Nadahalli, and Roger Wattenhofer. Timelocked bribes. *IACR Cryptol. ePrint Arch.*, 2020:774, 2020.

[KRvW17] Tommy Koens, Coen Ramaekers, and Cees van Wijk. Efficient zero-knowledge range proofs in ethereum. https://www.ingwb.com/media/2667860/zero-knowledge-range-proofs.pdf, 2017.

[LF20] Dough Leith and Stephen Farrell. Testing apps for COVID-19 tracing (TACT). https://down.dsg.cs.tcd.ie/tact/, 2020. Accessed: 2020-08-23.

[LHML20] Franck Legendre, Mathias Humbert, Alain Mermoud, and Vincent Lenders. Contact tracing: An overview of technologies and cyber risks. *CoRR*, abs/2007.02806, 2020.

[LK17] Kevin Liao and Jonathan Katz. Incentivizing blockchain forks via whale transactions. In *Financial Cryptography*, 2017.

[MHM18] Patrick McCorry, Alexander Hicks, and Sarah Meiklejohn. Smart contracts for bribing miners. In *Financial Cryptography*, pages 3–18, 2018.

[oPH20a] Swiss Federal Office of Public Health. New coronavirus: Swisscovid app and contact tracing. https://www.bag.admin.ch/bag/en/home/krankheiten/ausbrueche-epidemien-pandemien/aktuelle-ausbrueche-epidemien/novel-cov/swisscovid-app-und-contact-tracing/datenschutzerklaerung-nutzungsbedingungen.html#-11360452, 2020. Accessed: 2020-08-23.

[oPH20b] Swiss Federal Office of Public Health. Swisscovid app: Data protection statement & conditions of use. https://www.bag.admin.ch/bag/en/home/krankheiten/ausbrueche-epidemien-pandemien/aktuelle-ausbrueche-epidemien/novel-cov/swisscovid-app-und-contact-tracing.html, 2020. Accessed: 2020-08-23.

[oPH20c] Swiss Federal Office of Public Health. Swisscovid app: Threshold change. https://www.bag.admin.ch/dam/bag/en/dokumente/cc/kom/swisscovid-anpassung-schwellenwerte.pdf.download.pdf/SwissCovid%20app_Threshold%20Change.pdf, 2020. Accessed: 2020-09-17.

[Pie20] Krzysztof Pietrzak. Delayed authentication: Preventing replay and relay attacks in private contact tracing. *IACR Cryptol. ePrint Arch.*, 2020:418, 2020.

[RGK20] Adam Krellenstein Rosario Gennaro and James Krellenstein. Exposure notification system may allow for large-scale voter suppression. https://www.melani.admin.ch/dam/melani/de/dokumente/2020/SwissCovid_Public_Security_Test_Current_Findings.pdf.download.pdf/SwissCovid_Public_Security_Test_Current_Findings.pdf, 2020. Accessed: 2020-08-23.

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, Heidelberg, August 1990.

[Sei20] Otto Seiskari. Contact Tracing BLE sniffer PoC. https://github.com/oseiskar/corona-sniffer, 2020. Accessed: 2020-06-02.

[ST20a] Semaphore's Team. Semaphore. https://semaphore.appliedzkp.org/, 2020. Accessed: 2020-09-15.

[ST20b] SwissCovid's Team. Swisscovid app gradle. https://github.com/DP-3T/dp3t-app-android-ch/blob/master/app/backend_certs.gradle, 2020. Accessed: 2020-09-22.

[ST20c] SwissCovid's Team. Swisscovid server controller. https://github.com/DP-3T/dp3t-sdk-backend/blob/a730a5b276591e5cc8b6c609e2b0ba29c6069eb6/dpppt-backend-sdk/dpppt-backend-sdk-ws/src/main/java/org/dpppt/backend/sdk/ws/controller/GaenController.java, 2020. Accessed: 2020-09-05.

[The03] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. www.openssl.org, 2003.

[TJS16] Jason Teutsch, Sanjay Jain, and Prateek Saxena. When cryptocurrencies mine their own business. In *Financial Cryptography*, pages 499–514, 2016.

[um20] Github user mt19937 64. Please publish the public key needed for verifying EN export files. https://github.com/immuni-app/immuni-documentation/issues/114, 2020. Accessed: 2020-08-23.

[up20] Github user pdehaye. Configuration of the apps changed overnight. https://github.com/immuni-app/immuni-documentation/issues/112, 2020. Accessed: 2020-09-17.

[Vau20a] Serge Vaudenay. Analysis of DP3T. *IACR Cryptol. ePrint Arch.*, 2020:399, 2020.

[Vau20b] Serge Vaudenay. Centralized or decentralized? the contact tracing dilemma. *IACR Cryptol. ePrint Arch.*, 2020:531, 2020.

[VTL17] Yaron Velner, Jason Teutsch, and Loi Luu. Smart contracts make bitcoin mining pools vulnerable. In *Financial Cryptography*, pages 298–316, 2017.

[VV20a] Serge Vaudenay and Martin Vuagnoux. Analysis of swisscovid. https://lasec.epfl.ch/people/vaudenay/swisscovid/swisscovid-ana.pdf, 2020. Accessed: 2020-08-23.

[VV20b] Serge Vaudenay and Martin Vuagnoux. The dark side of swisscovid. https://lasec.epfl.ch/people/vaudenay/swisscovid.html, 2020. Accessed: 2020-08-23.

[VV20c] Serge Vaudenay and Martin Vuagnoux. Lessons from swisscovid. https://lasec.epfl.ch/people/vaudenay/swisscovid/lessons-from-swisscovid.pdf, 2020. Accessed: 2020-09-07.

[VV20d] Serge Vaudenay and Martin Vuagnoux. Little thumb attack on swisscovid. https://vimeo.com/453948863, 2020. Accessed: 2020-09-07.

[Wet] Dirk Wetter. testssl.sh. https://testssl.sh/.

[Wil18] Zachary J. Williamson. Aztec. https://github.com/AztecProtocol/AZTEC/blob/master/AZTEC.pdf, 2018. Accessed: 2020-09-15.

[Yan19] H. Yang. *EC Cryptography Tutorials - Herong's Tutorial Examples.* Herong's Tutorial Examples. Herong Yang, 2019.

[ZCC+16] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 270–282. ACM Press, October 2016.

[ZMM+19] Fan Zhang, Sai Krishna Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. DECO: liberating web data using decentralized oracles for TLS. *CoRR*, abs/1909.00938, 2019.

[ZT20a] ZkDAI's Team. Zkdai. https://github.com/atvanguard/ethsingapore-zk-dai, 2020. Accessed: 2020-09-15.

[ZT20b] ZoKrates's Team. Zokrates. https://zokrates.github.io/, 2020. Accessed: 2020-09-15.

A Tools

A.1 MACs and Signature Schemes

A *Message-Authentication Code* consists of a tuple of algorithms $(\texttt{Gen}, \texttt{Tag}, \texttt{Ver})$ such that

$\texttt{Gen}(1^\lambda)$: Takes as an input the security parameter and outputs a key k in the key space $\mathcal{K}$.

$\texttt{Tag}(m; k)$: Takes as an input a message m in the message space $\mathcal{M}$ and a key k, and outputs a tag θ.

$\texttt{Ver}(m, \theta; k)$: Takes as an input a message m and a key k, and outputs 1 iff θ is a correct tag on m under key k.

It must satisfy the following properties:

- **Completeness**: The probability that $\texttt{Ver}((m, \theta); k)$ outputs 1 for an honestly generated tag $\theta \leftarrow \texttt{Tag}(m; k)$ is 1.
- **Unforgeability**: The probability that an adversary, knowing only challenge message m^* and having access to an oracle giving back tags θ_i on messages $m_i \neq m^*$ (for all $i \in [n]$ with n polynomially bounded in the security parameter), outputs a pair (m^*, θ^*) such that $\texttt{Ver}(m^*, \theta^*; k) = 1$ is negligible.

A *Signature Scheme* consists of a set of algorithms $(\texttt{Gen}, \texttt{Sign}, \texttt{Ver})$, such that

$\texttt{Gen}(1^\lambda)$: Takes as an input the security parameter and outputs a pair $(\mathsf{sk}, \mathsf{vk})$ sampled from the key space, where sk is the signing key and vk the verification key.

$\texttt{Sign}(m; \mathsf{sk})$: Takes as an input a message m in the message space $\mathcal{M}$ and a signing key sk, and outputs a signature σ_m on that message.

$\texttt{Ver}(m, \sigma; \mathsf{vk})$: Takes as an input a pair (m, σ) and the verification key vk, and outputs 1 if the signature σ correctly verifies under vk.

It must satisfy the following properties:

- **Completeness**: The probability that $\texttt{Ver}((m, \sigma); \mathsf{vk})$ outputs 1 for an honestly generated signature $\sigma \leftarrow \texttt{Sign}(m; \mathsf{sk})$ is 1.
- **Unforgeability**: The probability that an adversary, knowing only the challenge message m^* and having access to an oracle giving back tags σ_i on messages $m_i \neq m^*$ (for all $i \in [n]$ with n polynomially bounded in the security parameter), outputs a pair (m^*, σ^*) such that $\texttt{Ver}(m^*, \sigma^*; \mathsf{vk}) = 1$ is negligible.

A.2 Public-Key Encryption Schemes

A Public-Key Encryption Scheme is a tuple of algorithms $(\texttt{Gen}, \texttt{Enc}, \texttt{Dec})$ such that

$\texttt{Gen}(1^\lambda)$: Takes as an input the security parameter, outputs a couple $(\mathsf{pk}, \mathsf{sk})$ of keys sampled in the key spaces.

$\texttt{Enc}(m; \mathsf{pk})$: Takes as an input a message m in the message space and a public key pk, and outputs the ciphertext c in the ciphertext space.

$\texttt{Dec}(c; \mathsf{sk})$: Takes as an input a ciphertext c and a secret key, and output a message m'.

A PKE scheme is CPA-Secure if the following properties are satisfied

- **Completeness** The probability that $m = m'$, where $m' \leftarrow \texttt{Dec}(c; \mathsf{sk})$ with $c \leftarrow\$ \texttt{Enc}(m; \mathsf{pk})$ for an honestly generated pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow\$ \texttt{Gen}(1^\lambda)$ is 1.
- **CPA-Security** The probability that an attacker, after choosing two messages (m_0, m_1), giving them to a challenger, and receiving back the encryption of one of the two (chosen by the challenger flipping a coin), can distinguish which of the two messages were encrypted, is negligible.

A.3 NIZK proofs

In a *zero-knowledge proof* system [GMR85] an entity $\mathcal{P}$, called prover, can prove to another entity, called verifier, that an NP-statement x is in some language $\mathcal{L}$ (i.e., exists at least a witness w such that the relation $\mathcal{R}_\mathcal{L}(x, w)$ for the language $\mathcal{L}$ is satisfied) without revealing a single bit of information on the witness used. Informally, the following properties must be satisfied by a zero-knowledge proof system:

- **Completeness:** The probability that an honest prover $\mathcal{P}$ (i.e. computing the proof by providing a valid (x, w) such that $\mathcal{R}_\mathcal{L}(x, w) = 1$) convinces the verifier $\mathcal{V}$ about the validity of the statement is 1.
- **Soundness:** The probability that a cheating prover convinces the verifier that a statement x is not in the language $\mathcal{L}$ is negligible.
- **Zero Knowledge:** If the statement x is true, the verifier learns no more information other than the fact that the statement is true. This concept is formalized by showing that there exists an efficient simulator that, given only the statement, can produce a protocol transcript that is indistinguishable from a real protocol execution.

A proof is said to be non-interactive when the interaction consists solely on a message sent by the prover to the verifier. A *zero-knowledge proof of knowledge* is a zero-knowledge proof where the prover shows that he actually *knows* a witness for the statement x and this is formalized by showing an efficient extractor that gives a witness in output. When we refer to NIZK proofs throughout the paper we usually intend NIZK-PoKs.

In the random oracle model both prover and verifier access to a cryptographic hash function that in the security proof is modelled as a random oracle. The simulator for the zero-knowledge property and the extractor for the proof of knowledge property have the power to program the random oracle.

B Useful Instantiations

B.1 The ECDSA scheme

The ECDSA signature scheme works over an elliptic curve of prime order n with generator P. The key generation algorithm works as follows. Let $d \in [1, n-1]$ be an uniformly random chosen integer, the public key is given by $Q = dP$ while the private key is d. We now describe the signature and verification algorithms. Let m be the message to sign, and $h(m)$ its hash.

$\mathsf{Sign}(m; d)$**:** Takes as an input a message m and the private key d, and performs the following steps.

- Sample a random $k \in [1, n-1]$
- Sets $R := (r_x, r_y) = kP$,
- $r := r_x \mod n$,
- $s := k^{-1}(h(m) + rd) \mod n$,
- and return (r, s).

$\mathsf{Verify}(m, (r, s); Q)$**:** Takes as an input a message m, a pair (r, s) and the public key Q. It performs the following steps.

- Calculates $v = h(m)P + rQ$ and
- if $sR = v$ then return 1;
- else returns 0.

B.2 Proving Knowledge of ECDSA signature

Consider a message m along with its ECDSA signature (r, s). $\mathcal{P}$ wants to prove he owns a valid signature (r, s) for a message m to a verifier $\mathcal{V}$, without letting $\mathcal{V}$ obtain such signature. Since r is perfectly simulatable, giving it away would obviously not allow a malicious $\mathcal{V}$ to derive s in order to compute a valid signature on m. After having revealed r, $\mathcal{P}$ could now prove in zero-knowledge that he knows s such that $\mathsf{Verify}(m, (r, s), Q)$ outputs 1 for given m, s and Q. By taking a look at the verification procedure (cfr., Section B.1) one can note that it basically consists of raising R, a value known to $\mathcal{V}$, to the witness s and checking if it is equal to v, which is also known to $\mathcal{V}$. Therefore it is evident that proving the above statement is equivalent to proving the knowledge of the discrete logarithm of v. This can be easily and efficiently done by using the Schnorr Σ-protocol [Sch90]. The protocol can be converted to a NIZK proof of knowledge in the random oracle model via the Fiat-Shamir heuristic [FS87]. In addition, one can make this proof non-transferable by giving in input to the random oracle an information which identifies the prover.

C Adding Seller's Privacy

As discussed in Section 2.2, using publicly posted TEKs is dangerous for the seller due to possible risks of incrimination. This could disincentivize the seller to utilize such smart contract mechanism. To guarantee seller's privacy, in all of our attacks we can enrich our playground by assuming the existence of a CPA-Secure PKE encryption scheme ($\mathtt{Gen}, \mathtt{Enc}, \mathtt{Dec}$) and a NIZK proof system. The proposed protocols can be modified as follows:

- When the buyer creates the smart contract, waits that a seller $\mathcal{P}$ is elected before providing his TEKs. When $\mathcal{P}$ is elected, $\mathcal{B}$ posts his TEKs encrypted with $\mathcal{P}$'s public key $\mathsf{pk}_{\mathcal{P}}$, by triggering an algorithm $\mathtt{SendBuyerTeks}(\mathbf{C}_{\mathcal{B}})$ where $\mathbf{C}_{\mathcal{B}} = (c_1, \ldots, c_n)$, with $c_i \leftarrow\$\ \mathtt{Enc}(t_i)$ for each $t_i \in \mathbf{T}_{\mathcal{B}}$. TEKs are pairs $t_i = (\mathsf{tek}_i, \mathsf{date}_i)$.
- When the signed TEKs list is available, the seller triggers $\mathtt{SendTeks}(\mathbf{T}, \sigma_T, \Pi, \tilde{\mathbf{T}})$, where $\mathbf{T} = (\tilde{t_1}, \ldots, \tilde{t_N})$ are the published TEKs, σ_T the corresponding signature, and $\Pi = (\pi_1, \ldots, \pi_n)$ is a sequence of proofs in which π_i is a NIZK proof that the prover knows $t_i \leftarrow \mathtt{Dec}(c_i; \mathsf{sk}_{\mathcal{P}})$ and that at least one element $\tilde{t_j}$ in a subset $\tilde{\mathbf{T}} \subseteq \mathbf{T}$ such that $|\tilde{\mathbf{T}}| > |\mathbf{T}_{\mathcal{B}}|$ is equal to t_i. The Smart Contract checks all the proofs, and if all of them verify, transfer the prize to the seller.

Now the only information that an external observer can deduce by looking at the proofs is that all the encrypted buyer's TEKs are indeed inside the list (or in a subset of them). To be sure that an observer cannot pinpoint the buyer's TEKs precisely, it is sufficient that the proofs use as a statement a subset of the published TEKs that contains at least one more TEK w.r.t the buyer's TEKs (proving on a subset and not on the entire list can be beneficial in terms of proof size and efficiency). The only harmful case is when the number of published keys matches with the number of the buyer's keys. We can argue that this condition happens quite rarely, considering that one external more key is sufficient to guarantee buyer's safety, and if GAEN recommendations are followed, a decent amount of keys should be present in the list.

D TLS Connections with Immuni and SwissCovid

In this section we show useful informations about TLS sessions established running on the client side the tools `openssl` [The03], and `testssl` [Wet] in order to connect to the TEKs upload servers of Immuni and SwissCovid.

Immuni. Using `testssl` to connect to `upload.immuni.gov.it` one can see that the server accepts TLS 1.2 connections only and the preferred ciphersuite is `ECDHE-RSA-AES256-GCM-SHA384, 256 bit ECDH (P-256)`. Moreover, one can also use `ECDHE-RSA-AES128-GCM-SHA256`.

Using `openssl` to connect to `upload.immuni.gov.it` one can see all the parameters of the established TLS session, including in particular a timeout of 7200 seconds (i.e., 2 hours) that is very large and thus beneficial for our attacks. For completeness we show here the content of the standard output (we replace some potentially identifying data by "...").

```
openssl s_client -connect upload.immuni.gov.it:443 -cipher ECDHE-RSA-AES256-GCM-SHA384

CONNECTED(00000003)
depth=2 C = IT, L = Milan, O = Actalis S.p.A./03358520967, CN = Actalis Authentication Root
CA
verify error:num=19:self signed certificate in certificate chain
```

```
verify return:1
depth=2 C = IT, L = Milan, O = Actalis S.p.A./03358520967, CN = Actalis Authentication Root
CA
verify return:1
depth=1 C = IT, ST = Bergamo, L = Ponte San Pietro, O = Actalis S.p.A./03358520967, CN =
Actalis Organization Validated Server CA G2
verify return:1
depth=0 C = IT, ST = ROMA, L = ROMA, O = Sogei S.p.A., OU = Server Sicuri, CN =
upload.immuni.gov.it
verify return:1
---
Certificate chain
 0 s:C = IT, ST = ROMA, L = ROMA, O = Sogei S.p.A., OU = Server Sicuri, CN =
upload.immuni.gov.it
   i:C = IT, ST = Bergamo, L = Ponte San Pietro, O = Actalis S.p.A./03358520967, CN =
Actalis Organization Validated Server CA G2
 1 s:C = IT, ST = Bergamo, L = Ponte San Pietro, O = Actalis S.p.A./03358520967, CN =
Actalis Organization Validated Server CA G2
   i:C = IT, L = Milan, O = Actalis S.p.A./03358520967, CN = Actalis Authentication Root CA
 2 s:C = IT, L = Milan, O = Actalis S.p.A./03358520967, CN = Actalis Authentication Root CA
   i:C = IT, L = Milan, O = Actalis S.p.A./03358520967, CN = Actalis Authentication Root CA
---
Server certificate
-----BEGIN CERTIFICATE-----
```

MIIGYzCCBUugAwIBAgIQRnoupUMoTHxpWMpARKffOTANBgkqhkiG9w0BAQsFADCB
lTELMAkGA1UEBhMCSVQxEDAOBgNVBAgMB0JlcmdhbW8xGTAXBgNVBAcMEFBvbnRl
IFNhbiBQaWV0cm8xIzAhBgNVBAoMGkFjdGFsaXMgUy5wLkEuLzAzMzU4NTIwOTY3
MTQwMgYDVQQDDCtBY3RhbGlzIE9yZ2FuaXphdGlvbiBWYWxpZGF0ZWQgU2VydmVy
IENBIEcyMB4XDTIwMDUxMzA3MTcyN1oXDTIxMDUxMzA3MTcyN1oweTELMAkGA1UE
BhMCSVQxDTALBgNVBAgMBFJPTUExDTALBgNVBAcMBFJPTUExFTATBgNVBAoMDFNv
Z2VpIFMucC5BLjEWMBQGA1UECwwNU2VydmVyIFNpY3VyaTEdMBsGA1UEAwwUdXBs
b2FkLmltbXVuaS5nb3YuaXQwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIB
AQC/2cqmDJieHJ8dMOzT/dlPpLGCAfWukniW5eFdNexZK3qxpESrrBm27OT6llLv
Oui/laRlqOHwg+2xycf1+aIGFiTO8dKuVyoCdJxCiQqCNF9dtMpgA69DgfYv/4o6
GwXvyx0PjQ/eF5+wJbjO1hIGYDm83J0WNbmHhqA8NzHrqP5Q554aNYrLnzXDVpdN
3I3Gdo/KlUkuH5RUutYhbVZand8uf069MFROzL1xifdHLVqCwYrNRkyc6BqyCVoV
c2f1TEQZ9T9OQxijXkMdXXwNkUXKS6O/SVtGUiUm2KgQ098XqOzEs6U/OaWVyFPt
YMSSh8hpT3bR3eaLjI2yMX/5AgMBAAGjggLIMIICxDAMBgNVHRMBAf8EAjAAMB8G
A1UdIwQYMBaAFGL+uyeKZETtaJZaWHmh21omrf+7MH4GCCsGAQUFBwEBBHIwcDA7
BggrBgEFBQcwAoYvaHR0cDovL2NhY2VydC5hY3RhbGlzLml0L2NlcnRzL2FjdGFs
aXMtYXV0aG92ZzIwMQYIKwYBBQUHMAGGJWh0dHA6Ly9vY3NwMDkuYWN0YWxpcy5p
dC9WQS9BVVRIT1YtRzIwHwYDVR0RBBgwFoIUdXBsb2FkLmltbXVuaS5nb3YuaXQw
UQYDVR0gBEowSDA8BgYrgR8BFAEwMjAwBggrBgEFBQcCARYkaHR0cHM6Ly93d3cu
YWN0YWxpcy5pdC9hcmVhLWRvd25sb2FkMAgGBmeBDAECAjAdBgNVHSUEFjAUBggr
BgEFBQcDAgYIKwYBBQUHAwEwSAYDVR0fBEEwPzA9oDugOYY3aHR0cDovL2NybDA5
LmFjdGFsaXMuaXQvUmVwb3NpdG9yeS9BVVRIT1YtRzIvZ2V0TGFzdENSTDAdBgNV
HQ4EFgQUFCzbrEIZXwf4JgUy4YCgmpuPKJcwDgYDVR0PAQH/BAQDAgWgMIIBBQYK
KwYBBAHWeQIEAgSB9gSB8wDxAHYARJRlLrDuzq/EQAfYqP4owNrmgr7YyzG1P9Mz
lrW2gagAAAFyDO5cFgAABAMARzBFAiBc/J6oayZGC43Uoec5S432UxCy/AmXaX2P
OgDXEUjJxwIhAKr6mS9OXWxe/wa599GmXLDOFYG7QFYYt3Hw2ef/7hk6AHcA9lyU
L9F3MCIUVBgIMJRWjuNNExkzv98MLyALzE7xZOMAAAFyDO5cVQAABAMASDBGAiEA
sdib2FsWyErV+T3IgJqnlw0quecJ8nlerqxHSi+jX+MCIQDySdDQ5ssmiu3pW9MY
60td+s/UOb6oIekdqZKCYTv1aDANBgkqhkiG9w0BAQsFAAOCAQEADODHgxyVPgl+
IOwRl6huoOiaFseBfR5dBHTyPa/axCCZxwtZNU8rkPPWfHp36e4iSbOHwEmMjAEr
h9lWR786ohUerN9EUd98Xais/RgJOuN1TZfQM72nmgw0hYciyOMyUmULUSbbPSDs
JL5zs3pn2E7oCoagdNS14kpp/LDGo8iwitTK7XYtdOu/SAv1k9WfLjY3tR+hNAfJ
R9FTxyRYNFDOaWlwMgDISAdS3WHU8yJ6QntFTpLQ3vEHvTgswe+pTFzz2Yxl3CFk
DOAbbJ7AcBWDbAr+9H6GDF8uQ7om5SVm0CIUqdQRlG6RMetdz+36QPWuNiCV4c4O
gP7cycp+/Q==

```
-----END CERTIFICATE-----
subject=C = IT, ST = ROMA, L = ROMA, O = Sogei S.p.A., OU = Server Sicuri, CN =
upload.immuni.gov.it

issuer=C = IT, ST = Bergamo, L = Ponte San Pietro, O = Actalis S.p.A./03358520967, CN =
Actalis Organization Validated Server CA G2

---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: RSA
Server Temp Key: ECDH, P-256, 256 bits
---
SSL handshake has read ... bytes and written ... bytes
Verification error: self signed certificate in certificate chain
---
New, TLSv1.2, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : ECDHE-RSA-AES256-GCM-SHA384
    Session-ID: ...
    Session-ID-ctx:
    Master-Key: ...
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    Start Time: ...
    Timeout   : 7200 (sec)
    Verify return code: 19 (self signed certificate in certificate chain)
    Extended master secret: no
---
```

SwissCovid. Using `testssl` to connect to `www.pt1.bfs.admin.ch` one can see that the server accepts TLS 1.2 connections only and accepts, interestingly, also the CBC-HMAC ciphersuite, therefore allowing a more efficient attack using DECO. We report here a text extracted from the standard output.

```
ECDHE-RSA-AES128-GCM-SHA256 ECDHE-RSA-AES256-GCM-SHA384
ECDHE-RSA-AES128-SHA256 ECDHE-RSA-AES256-SHA384
ECDHE-RSA-AES128-SHA

Cipher Suite Name (OpenSSL)   KeyExch.   Encryption  Bits  Cipher Suite Name (IANA/RFC)
--------------------------------------------------------------------------------------------
ECDHE-RSA-AES256-GCM-SHA384   ECDH 384   AESGCM   256   TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
ECDHE-RSA-AES256-SHA384       ECDH 384   AES      256   TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
ECDHE-RSA-AES128-GCM-SHA256   ECDH 384   AESGCM   128   TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
ECDHE-RSA-AES128-SHA256       ECDH 384   AES      128   TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
ECDHE-RSA-AES128-SHA          ECDH 384   AES      128   TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
```

Using `openssl` to connect to `www.pt1.bfs.admin.ch` with the CBC-HMAC ciphersuite,

one can see all the parameters of the established TLS session, including in particular a timeout of 7200 seconds (i.e., 2 hours), that is very large and thus beneficial for our attacks. For completeness we show here the standard output (we replace some potentially identifying data by "...").

```
openssl s_client -connect www.pt1.bfs.admin.ch:443 -cipher ECDHE-RSA-AES128-SHA

CONNECTED(00000003)
depth=2 C = BM, O = QuoVadis Limited, CN = QuoVadis Root CA 2 G3
verify error:num=19:self signed certificate in certificate chain
verify return:1
depth=2 C = BM, O = QuoVadis Limited, CN = QuoVadis Root CA 2 G3
verify return:1
depth=1 C = BM, O = QuoVadis Limited, CN = QuoVadis Global SSL ICA G3
verify return:1
depth=0 C = CH, ST = Bern, L = Bern, O = Bundesamt fuer Informatik und Telekommunikation
(BIT), OU = Swiss Government PKI, CN = www.pt1.bfs.admin.ch
verify return:1
---
Certificate chain
 0 s:C = CH, ST = Bern, L = Bern, O = Bundesamt fuer Informatik und Telekommunikation
(BIT), OU = Swiss Government PKI, CN = www.pt1.bfs.admin.ch
   i:C = BM, O = QuoVadis Limited, CN = QuoVadis Global SSL ICA G3
 1 s:C = BM, O = QuoVadis Limited, CN = QuoVadis Global SSL ICA G3
   i:C = BM, O = QuoVadis Limited, CN = QuoVadis Root CA 2 G3
 2 s:C = BM, O = QuoVadis Limited, CN = QuoVadis Root CA 2 G3
   i:C = BM, O = QuoVadis Limited, CN = QuoVadis Root CA 2 G3
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIHrDCCBZSgAwIBAgIUfE2HPqjnQrTf9FnR+q9p2lVpz3cwDQYJKoZIhvcNAQEL
BQAwTTELMAkGA1UEBhMCQk0xGTAXBgNVBAoTEFF1b1ZhZGlzIExpbWl0ZWQxIzAh
BgNVBAMTGlF1b1ZhZGlzIEdsb2JhbCBTU0wgSUNBIEczMB4XDTIwMDQyMTEwNDMw
OVoXDTIyMDQyMTEwNTMwMFowgakxCzAJBgNVBAYTAkNIMQ0wCwYDVQQIDARCZXJu
MQ0wCwYDVQQHDARCZXJuMT4wPAYDVQQKDDVCdW5kZXNhbXQgZnVlciBJbmZvcm1h
dGlrIHVuZCBUZWxla29tbXVuaWthdGlvbiAoQklUKTEdMBsGA1UECwwUU3dpc3Mg
R292ZXJubWVudCBQS0kxHTAbBgNVBAMMFHd3dy5wdDEuYmZzLmFkbWluLmNoMIIB
IjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA35huk0exPmA6JIyWxvxJWWvG
cB4AkeLloyKTf0nFr5NA4QxIhX8YmksDLL+HfkY+4Kf58PuWT00li0OKpF4h9b4E
q5sOLFTJEkCmwLoZuP5E7dKFs0tXLfR/mH/5CqQAdLW8Tnhn2ZJ5YFvSDzjSOYuX
irRr5N3Y/FKrHa7ggYJKSdjvT25BIjtkimgEORAzqEwrgxgFgJ7rRqVVZ9G8I5q7
y3RIwYmJ4Qf/aF3R1iyYUxbcMQvk8G1sS/JEJ+MBALwgWXVNhswQSi0PcVtlSpNk
TGNB73SBbrzGEdozZqYQnoS378w0wu0iepY+K/hZx1D/euIe7CCxBG7XGKs+nQID
AQABo4IDJTCCAyEwCQYDVR0TBAIwADAfBgNVHSMEGDAWgBSzEom1qUs1vBUA8IDp
2HiH8RN8djBzBggrBgEFBQcBAQRnMGUwNwYIKwYBBQUHMAKGK2h0dHA6Ly90cnVz
dC5xdW92YWRpc2dsb2JhbC5jb20vcXZzc2xnMy5jcnQwKgYIKwYBBQUHMAGGHmh0
dHA6Ly9vY3NwLnF1b3ZhZGlzZ2xvYmFsLmNvbTAfBgNVHREEGDAWghR3d3cucHQx
LmJmcy5hZG1pbi5jaDBRBgNVHSAESjBIMEYGDCsGAQQBvlgAAmQBATA2MDQGCCsG
AQUFBwIBFihodHRwOi8vd3d3LnF1b3ZhZGlzZ2xvYmFsLmNvbS9yZXBvc2l0b3J5
MB0GA1UdJQQWMBQGCCsGAQUFBwMCBggrBgEFBQcDATA6BgNVHR8EMzAxMC+gLaAr
hilodHRwOi8vY3JsLnF1b3ZhZGlzZ2xvYmFsLmNvbS9xdnNzbGczLmNybDAdBgNV
HQ4EFgQU2F1pjrbfsqvpeWohTrbIn/4jRRQwDgYDVR0PAQH/BAQDAgWgMIIBfgYK
KwYBBAHWeQIEAgSCAW4EggFqAWgAdwCkuQmQtBhYFIe7E6LMZ3AKPDWYBPkb37jj
d80OyA3cEAAAAXGcXsluAAAEAwBIMEYCIQDJlsbMecVS6415SAqAI6ZqmRQLJq9M
U1dWJb/8fKJ2+gIhAJSksFLfhwuihMum/cONG0Bw1SnPODhBNLW1zAGeQnggAHUA
VhQGmi/XwuzT9eG9RLI+x0Z2ubyZEVzA75SYVdaJ0N0AAAFxnF7JcgAABAMARjBE
AiBNZUPkn8ArJVVnyRlthxUILagDylovYL393EbkQKXimwIgfdrGHf5n6Sjla2CC
```

```
P96wGdbOTls19hDMlYZK3W+eIQcAdgBvU3asMfAxGdiZAKRRFf93FRwR2QLBACkG
jbIImjfZEwAAAXGcXsncAAAEAwBHMEUCIGgZ4z5MgpIlcLvBkge/BEqJ/7sYT3ze
IXMrVizfwj2TAiEAyJYfJ6D9AYxRyvDcTwnWfHXers3SNbi7sOPuX5lkWsMwDQYJ
KoZIhvcNAQELBQADggIBAC3O3iZDQYcKZ+DyAx4HSzwLpIa5yMiungbkmQuN7RYO
4OpPIiADc/V/P/x+cDSuttJa8eoUq9zXEA9VW+ETFOWszf5WE31+MjasmTqDyjqV
tNyrjAGACPhbH3J9ydGQX3SqdrGNwFiBRwTvxPqukFu3+JIoRpYMzwXfbRnig1fW
R+creYRgloizGYu/M4gqV8LBwE/k7plrwTwsA8BhijhcR5asC+htRSB2SaS+teNO
ski5EJ4ajcv78vkN9y+BFKMcq3Cb5jCjoCleUoMm/BVoQNsOZAcLJmQ3VHVeeY/S
drI9OzODQ8dSyCZKm5KDMBS2inOljLyPinbtk7JZWLQDetbW1BsjaC98BNeriYQ1
OaavGrOTPFrC9NWAB5ze3342LsTPZwiIEeGhB4AGsVTO3oYRi3yc52r22W4OmdcH
f+fRgCrqddoKdPgGGp8+p9+IsWConFjzEQpN5iJRJlzorCw8nvvuKkG2I+mj9hMK
WA3/r5mOC+/ZUw+rYLpLstbCoHDBymAFFOb6P66tMl3JSvb5/Rjnhrhmtccq7o5k
Qfmr2tGrNs+8cdrVN9efn4es3OfljVGP98YNtVHz7rC/nicXU5eLxHSHna4iOwlU
wpuX5DOrnF35YOYP6eJsJS/p7y+k3L3N/iavLOjOLVIILSlTJzOZgDSbK4tG6PNk
-----END CERTIFICATE-----
subject=C = CH, ST = Bern, L = Bern, O = Bundesamt fuer Informatik und Telekommunikation
(BIT), OU = Swiss Government PKI, CN = www.pt1.bfs.admin.ch

issuer=C = BM, O = QuoVadis Limited, CN = QuoVadis Global SSL ICA G3

---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: RSA
Server Temp Key: ECDH, P-256, 256 bits
---
SSL handshake has read ... bytes and written ... bytes
Verification error: self signed certificate in certificate chain
---
New, TLSv1.0, Cipher is ECDHE-RSA-AES128-SHA
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : ECDHE-RSA-AES128-SHA
    Session-ID: ...
    Session-ID-ctx:
    Master-Key: ...
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    Start Time: ...
    Timeout   : 7200 (sec)
    Verify return code: 19 (self signed certificate in certificate chain)
    Extended master secret: no
---
```