WHY CONIKS FAILED

CSCI 388: Cryptography

Louise Xu
Niels Christoffersen
Professor Chanathip Namprempre
Reed College

May 2023

Abstract—**We explore the security goals and flaws of CONIKS, a security model that helps provide end-to-end encrypted communication systems with decentralized public key directories. CONIKS assumes a malicious key service provider and uses Merkle trees to publish privacy-preserving digests that can be audited efficiently. However, CONIKS implements two security policies, a default policy and a strict policy, which tradeoff usability and security. The default policy doesn't require key change requests to be signed, which makes key recovery easy, but leads to false positives or undetected malicious activity, while the strict policy requires signed requests, which provides cryptographic proof of any malicious activity but renders key recovery difficult for users. The lack of a definitive working key change policy is one of the main barriers that hinder the implementation of CONIKS in the industry. Nonetheless, the developers of CONIKS are still hopeful for a deployable version in the future.**

I. Introduction

End-to-end encryption is a secure communication technique that ensures only the sender and the receiver can access the message content. This is achieved by encryption the message on the sender's device before sending, and decrypting it on the receiver's device after receiving. This helps guarantee that the transmitted data would be unreadable to third parties during the transmission who don't have the necessary keys.

End-to-end encryption is realized through public key encryption. Suppose Alice wants to send Bob an email, then Alice needs to encrypt the message with Bob's public key so that Bob can decrypt it using his secret key. This requires Alice to reliably obtain the public key corresponding to Bob. Many large communication services such as Apple iMessage and WhatsApp have deployed end-to-end encryption rely on a centralized directory of public keys maintained by the service provider. In this case, Alice would query the service provider for Bob's public key and trust that it gives the key that is bound to Bob. These centralized key servers remain vulnerable to technical compromise [1,2], and legal or extralegal pressure for access by surveillance agencies or others.

Assuming the service provider to be the malicious adversary, Melera et al. propose CONIKS, a security model with decentralized service providers that can be audited efficiently and preserves the privacy of the public keys stored in the directories. CONIKS builds on transparency log solutions used for web server certificates and uses similar tools such as Merkle trees to provide publicly auditable yet privacy-preserving digests. However, CONIKS targets distinct security objectives from those of transparency logs, namely preventing non-equivocation and detecting spurious key changes. A provider is said to be equivocating if it returns a different key binding than the actual one represented in the key directory. In order to conceal its misbehavior, the provider may target this equivocation to specific entities only, and return the correct key binding to others. In other words, an equivocating provider would be presenting diverging views of the key bindings to different users. As for malicious key changes inserted by the provider, CONIKS aims to rapidly detect this and alert the user.

CONIKS relies on two primary key change policies that dictate how users can change, replace, or recover their keys. The first is the default key change policy, which was designed for increased usability. This policy does not require that key changes are signed by the public key of the user, making it easier for the user to change their key, but also less secure. The second is the strict key change policy, which requires that all new keys are signed by the private key of the user. This is more difficult for users to operate, but provides much more security and cryptographic proof of malicious activity.

However, these two key change policies were one of the main reasons why CONIKS was not adopted by any company. One of the main problems with the default key change was that key recovery was unsigned, meaning that if a user lost their key, then requested to recover it, there was no proof of who was requesting the key recovery. This would lead to false positives of malicious activity when it did not occur, or undetected malicious activity when it did. The main problem with the strict key change policy also revolved around key recovery, where if a user ever lost their private key, they would never be able to recover it as it was stored as cryptographically hashed data. The lack of a single definitive working key change policy was seen as a main barrier to its adoption in the industry.

We provide an overview of the public key infrastructure and where CONIKS situates in section 2, then introduce how CONIKS works in section 3, and finally illustrate the flaws with CONIKS key change policy in section 4.

II. Literature Review

Public Key Infrastructure (PKI) refers to technologies used to secure communication over the Internet. Most works of literature focus on what CONIKS refers to as the Web PKI, which CONIKS distinguishes from PKI for end-user key verification.

Web PKI is used primarily for secure communication between web browsers and web servers, typically using the SSL/TLS protocol. In web PKI, a web server presents a digital certificate to the web browser, which is issued by a trusted third-party Certificate Authority (CA) and contains information about the server's identity and public key. This digital certificate is signed by the CA's secret key and can be authenticated by the browser which has the CA public key. The web browser then uses the decrypted server public key to establish a secure connection with the web server, encrypting all data transmitted between the two. Web PKI relies on a hierarchy of trusted CAs to issue and manage digital certificates, with many browsers having common root CA public keys pre-installed.

However, there have been concerns about the security and trustworthiness of some CAs, leading to the development of alternative approaches such as Certificate Transparency (CT) logs. As Kales et al. nicely summarize, web clients need to ensure that log servers do not hand out promises to include the certificate in the log without actually doing so [3]. Thus to combat misbehaving log servers, web clients act as auditors and verify that any certificates they receive are actually publicly logged. Thus CT logs provide a publicly available record of all digital certificates issued by CAs, allowing researchers and others to monitor for potentially fraudulent certificates. Specifically, the log servers provide auditable digests of bindings in the form of Merkle tree roots [4]. Nevertheless, users directly contacting the CT log servers to retrieve inclusion proofs and verify the certificate inclusion creates another privacy problem because the users' browsing activities could be recorded by the log server owner.

PKI for end-user key verification, on the other hand, is used to establish trust between individual users, rather than between a web server and a web browser. This approach is used in various applications, such as secure messaging and file sharing. In end-to-end encryption setting, data is encrypted by the sender before being transmitted and then decrypted by the recipient after being received. Hence the sender needs the public key of the recipient in order to correctly encrypt the message. This requires that the sender can correctly obtain the public key that is associated with the sender.

A naive approach is to have users manually exchange keys through a trusted channel, but this approach lacks scalability. In PGP email encryption [5,6], for instance, verifying keys manually has been proven to be difficult to use and prone to mistakes [7,8]. A common but imperfect approach is to have users upload and store their public key bindings in a public key directory. Many services that strive to provide end-to-end encryption nowadays, such as Apple iMessage and Whatsapp, rely on such a centralized directory of public keys maintained by the service provider [9,10]. The centralized scheme requires the users to trust that the service providers provide the correct key binding. However, these key providers may return the incorrect key if acting maliciously or under coercion. Therefore, CONIKS proposes a key verification scheme with decentralized service providers.

CONIKS builds on transparency log proposals for web server certificates but differs in many ways to better suit the assumptions of end-user key verification. Specifically, many protocols In web PKI aim to validate the existence of a CA in the log directory, so the contents of the directory are publicly available. Even though Merkle tree hash was used, it was only to provide efficient authentication of a certain certificate. However, in an end-user key verification setting, many providers would like to keep their users' names and public keys private. Secondly, these schemes rely to varying degrees on third-party monitors interested in ensuring the security of web PKI on the whole, since monitoring the certificates or bindings issued for a single domain or user requires tracking the entire log. However, there are hundreds of thousands of email providers and communication applications, most of which are too small to be monitored by independent parties.

CONIKS addresses these two problems with two improvements over the traditional PKI solutions: efficient monitoring and privacy-preserving key directories. Efficient monitoring means that it is feasible for a single client to monitor its binding without tracking the entire log history, and it is achieved through Merkle trees that allow users to monitor only their own entry without needing to rely on third parties to perform expensive monitoring of the entire tree. Privacy-preserving key directories mean that clients may only query for individual usernames and the response for any individual queries leaks no information about which other users exist or what key data is mapped to their username. A key observation to make is that privacy-preserving here does not refer to the same privacy concern that Web PKI is trying to address. Instead, CONIKS strives to preserve the privacy of other users from the querying user, but it does not tackle the problem of leaking query information to the server itself.

Another similarity to CONIKS that all CT log solutions share is that they reliably detect but do not prevent malicious activity within the PKI. Therefore, many security concerns for transparency log PKI solutions are still highly relevant to CONIKS, and we will be discussing some of them in our paper.

III. Introduction to CONIKS

A. *Consistency in key directories*

The service providers, also referred to as identity providers by CONIKS, issue authoritative name-to-key bindings within their namespaces, such as the namespace foo.com that binds the name alice@foo.com to one or more public keys associated with Alice. Furthermore, CONIKS ensures that users can automatically verify the consistency of these bindings, meaning anyone can confirm that this is the same binding for alice@foo.com observed by all parties. However, automating the stronger correctness property of bindings, which requires users to verify that the keys bound to alice@foo.com are genuinely controlled by Alice, is impractical. Instead, CONIKS relies on Alice to periodically monitor this binding and detect if it does not represent the keys that Alice actually controls.

B. *Participating parties in the CONIKS ecosystem*

Identity providers. The identity provider runs a CONIKS server that manages separate namespaces and issues authoritative name-to-key bindings within the namespaces. CONIKS assumes an ecosystem of multiple identity providers that each hold its own storage of the key bindings. The identity providers will periodically publish digests, which are auditable proofs of their bindings (see the section on STR), and provide proof that a given binding is represented in the digest when they respond to key requests (see the section on Merkle tree). CONIKS also assumes that a separate PKI system distributes the provider's public keys to clients.

Clients. The client refers to the client CONIKS software is run by users to query the public keys of other clients whom they wish to communicate with. The clients will monitor the consistency of their own key bindings periodically. CONIKS assumes that the clients behave well since the problem of compromised clients is not the focus of CONIKS. CONIKS also assumes that the clients have network access so that they can whistleblow after detecting misbehavior by an identity provider.

Auditors. Auditors track the chain of signed "snapshots" of the key directory. Auditors gossip with other auditors to ensure global consistency across all key directories. In fact, CONIKS clients can serve as auditors for their own identity providers, although third-party auditors are also possible.

Users. Users run the client CONIKS software to communicate with other users. Specifically, CONIKS defines two types of users: default users and strict users corresponding to default and strict security policies, which have different tradeoffs of security and privacy against usability.

C. Default vs Strict Users

The security policy consists of two parts: a key lookup policy that determines the visibility of a user's public keys in the identity provider, and a key change policy where the user wants to change its public key in the event that the previous device was lost or destroyed.

Key Lookup Policy: In the default lookup policy, a user's public key is not encrypted on the CONIKS server, while in the strict lookup policy, it is encrypted using a symmetric encryption key known only to the binding's owner or other chosen users. For example, if Alice follows the default lookup policy, then anyone who knows Alice's name alice@foo.com can look up and obtain her keys from her foo.com's directory. If Alice follows the strict lookup policy, then only users that Alice chose to share her symmetric key with can learn her public keys. The default policy provides more intuitive interactions because it allows any user to retrieve the public keys associated with a specific name without requiring explicit permission. On the other hand, the strict policy provides stronger privacy but requires additional action to distribute the symmetric encryption key to selected users.

Key Change Policy: The default key change policy allows for key changes to be made without requiring a cryptographic signing of the request with a secret key controlled by the user. Since unsigned key change statements are accepted, this policy makes it easy for clients who have lost their previous keys to reclaim their binding with a new key. However, it also creates an opportunity for identity providers to maliciously change a user's key.

For instance, if Alice chooses the default key change policy, her identity provider foo.com accepts any key change statement where the new key is signed by the previous key or is unsigned. This means that if Alice loses her previous key, she can still update her binding with a new key without authentication. The identity provider will update Alice's binding accordingly and include the key change statement, so any other clients such as Bob can see that a key change has been performed recently. However, if the identity provider maliciously changes Alice's key and falsely claims that Alice requested the operation, there is no way for Bob to distinguish this attack from an actual unauthenticated key change by Alice. Under this policy, only Alice can be certain that she has not requested the new key. As a result, CONIKS suggests that other clients who see an unauthenticated key change should notify Alice, making surreptitious key changes risky for identity providers.

In contrast, the strict policy mandates that all key change statements be signed with the previous key that is being changed. This policy ensures that any unsigned or incorrectly signed key change request will be rejected when observed by clients. Therefore, Alice's client can immediately detect a spurious key change when monitoring her bindings, and malicious bindings will not be accepted by other users. Moreover, Alice will have cryptographic proof of malicious behavior from the server (see the section in Attack for more detail). However, if Alice loses all her previous keys, there is no way to register a new key with the identity provider, and she will forfeit her name. Thus, the strict key change policy provides more security by forcing authentication for every key change, but at the expense of difficult recovery after key loss.

D. Merkle Trees

To publish a digest, the identity provider assembles all its bindings into a data structure called the Merkle tree. As in Fig. 1, The hashes of the bindings are represented by the bottom leaf nodes, and a binary tree is constructed by continuous pairwise hashing of two leaf nodes. Specifically, two leaf nodes are concatenated and then hashed to create a parent node. The top root of the tree is the digest that the identity provider publishes. It is a cumulative hash on the entire tree since a change in any bottom leaf node would result in a change in the root hash.

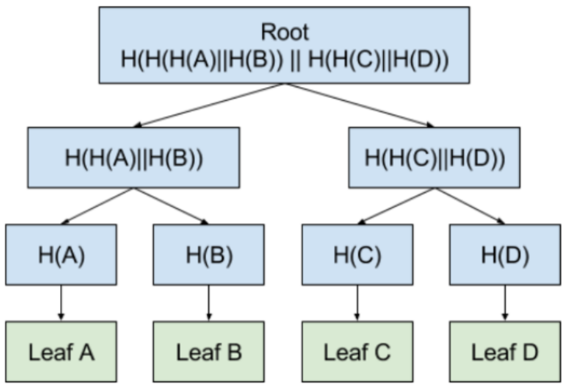

Fig. 1. Diagram of Merkle tree structure [11].

Authentication Path

Merkle tree is a useful data structure that enables efficient membership testing. Suppose a client requests for a key binding for B, and the identity provider responds with Leaf B. In order to prove to the client that leaf B is indeed in the Merkle tree stored in the server, the provider additionally sends the Root and an authentication path consisting of a set of adjacent intermediate nodes in the Merkle tree. In this example, the authentication path would be H(A), and H(H(C)||H(D)). The client, which has the same hash function, can then hash the received Leaf B, concatenate the result with H(A), and hash again to recompute the intermediate node H(H(A)||H(B)). The client can do so iteratively to recompute the root of the Merkle tree. If the recomputed value matches with the received Root value, then the client is convinced that the received key binding does indeed exist in the Merkle tree in the correct position.

STR and Non-equivocation

Since the Merkle tree root hash changes every time the key directory is updated, the identity providers need to publish the root hashes at regular intervals, or epochs. At each epoch, the STR consists of the current Merkle tree root, the hash of the previous STR, and other metadata and is signed by the provider's private key. Therefore, instead of sending the root hash directly to the client, the provider would send the STR. The client is assumed to have the provider's public key, so the client can decrypt the STR and read the current root hash value.

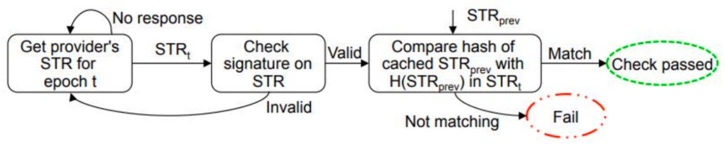

Fig. 2. The auditor first ensures that the provider correctly signed the STR before checking whether the embedded hash of the previous epoch's STR matches what the auditor saw previously. If they do not match, the provider has generated a fork in its STR history.

By including the previous STR's hash as part of each STR, the provider essentially commits to a single, linear history of digests. Suppose the client also wants to verify that the received root value is correct, then the client may act as an auditor and verify that the received STR does belong to the linear STR history. The client can check by obtaining the previous STR value from memory, hashing it, and comparing it with the received hash of the previous STR provided (A more detailed procedure is described in Fig. 2). Hence this linear chain is important in preventing equivocation, because if a malicious provider wants to equivocate for a key binding, then it must equivocate for the root hash, hence it must equivocate for the STR. The provider must create a fork in its linear history and maintain the forked branch for the rest of the time, otherwise, the client may detect an equivocation immediately because the hash of the previous STR does not match.

Nonetheless, in case the equivocation happened early in the STR history, verifying that the immediate history is linear may not be enough. Therefore, the client has a second method of checking, which is to perform STR comparisons with other auditors. Once a client has verified the provider's STR history is linear, the client queries one or more CONIKS auditors at random. The client then compares the other auditor's observed STR with the STR which the provider directly presented it. The client may repeat this cross-verification process with different auditors as desired to increase confidence. If a malicious CONIKS server returns the incorrect keys to certain users and returns the correct keys to others, this method of cross-verification may expose such malicious behavior to the client. The randomness of the selection of auditors is also critical because it prevents the case of "attack bubbles," wherein a directory equivocates to all users that are likely to contact each other [11].

IV. Attacks Against CONIKS and Attempted Detection and Prevention

CONIKS is proposed to be a PKI scheme that, unlike most traditional key directory PKIs, detects key tampering using a transparent and decentralized key authentication system. However, in this section, we will explore attacks that are used against PKIs, and determine if CONIKS is secure against them and if it is able to detect key tampering with cryptographic proof.

The first response against key change attacks that CONIKS provides is key lookup. This means that if Alice wants to message Bob, to verify they are talking to the correct receiver, Alice's client queries Bob's provider. The provider will return the STR, Bob's key binding, and the authentication path that proves the binding is part of the tree. However, it is possible that Bob's key binding has been modified or tampered with. We will explore how CONIKS can or cannot protect against this. First, Alice's client will check the authentication path with the key binding to the STR, and if this is determined invalid, then Alice's client will not trust the key binding. If the authentication path is determined valid with the STR, then the client will audit the STR by comparing it with the STRs of randomly selected identity providers and verifying correctness. If there is no distinction between the STR from Bob's provider and from the randomly selected clients, then Alice trusts Bob's key binding. However, this form of authentication means that Alice's client has no way of independently verifying the correctness of Bob's key, meaning that an adversarial identity provider could send the wrong key to Alice. Alice then relies on Bob's client to whistleblow if they discover an incorrect key binding.

The second response is key monitoring. Clients in CONIKS should be regularly monitoring their user's key bindings. This is done by the client storing the user's previous key, and then querying the identity provider for the user's key and comparing them to determine that they are the same. The authentication path to the key is also verified during key monitoring using the STR. The users in CONIKS also have the choice between using the default or strict key change protocols. The difference in key change protocols does not affect whether or not key monitoring detects an attack, as it always will. However, if a user uses the default key change protocol, the user's client cannot determine who the likely attacker was as the key was not signed. If a user is using the strict key change protocol this means their key is signed by the previous key's associated private key. Then if the new key is invalid but the signature is correct then the client does not know who tampered with the key. However, if the signature is invalid, then the client can have high confidence that it is the identity provider who tampered with the key. Another advantage of Alice using the strict key change protocol is that, when Bob's client looks up Alice's key, Bob's client can verify the strict-policy key changes by checking the signature on the key change request. Therefore, even if Alice's provider is malicious, Bob will be able to detect this. However for Alice to realize this, the client of Bob must whistleblow.

Whistleblowing is a necessary component of CONIKS, as it allows the detection of malicious activity by one client to be known by all participating parties. However, in the CONIKS paper, they declare: "we leave the complete specification of a whistleblowing protocol for future work" [12]. Still, it remains an integral part of the CONIKS system, as users and their clients rely on other clients providing cryptographic proof of malicious activity, whether that be in the form of an incorrect signature on a key, or an invalid STR.

If a client is monitoring its user's key and receives a response that is valid, but the key is incorrect, then the client must whistleblow to all other parties that the key is not to be trusted. However, this cannot be done by simply sending a message that the key is not to be trusted, as this could be spoofed by an attacker, rendering a potentially valid key incorrect. This means that when whistleblowing, the client must sign the message with their private key. However, this is not always useful or easy for CONIKS clients. Assuming the strict key change policy, if the user's secret key is known to the client, then whistleblowing is trivial as they just sign the message. However, if the secret key is unknown or compromised, then there is no way for the client to sign the whistleblowing message. This is one of the main problems with CONIKS' strict key policy, even outside of whistleblowing. If the user's secret key is lost, there is no way to recover it. This is extremely difficult for usability, as users will often forget passwords, and in this case, it would result in potential impersonation as well as loss of data, both without any way of prevention.

There are also difficulties with using the default key change policy. In a 2016 article written by CONIKS co-founder Marcela Melara outlining the difficulties with the protocol and the reasons why it wasn't adopted, she describes one of the issues with the default key change policy. This was the problem of key recovery when a user lost their key and must replace it. "We designed a default account recovery mechanism for CONIKS: unauthorized key changes. However, this mechanism undermines users' security since it doesn't leave cryptographic evidence and other app clients have no way of distinguishing account recovery from a compromised account. The engineers viewed developing a more secure account recovery mechanism, in which it's unambiguously clear who initiated the recovery, as one of the main barriers to deploying CONIKS [13]. The issue with unauthorized key changes is that the clients are unable to distinguish between an attempted malicious key recovery and an attempted recovery by the user themselves. This means that there are major issues with both the default and strict key change policies, and these were some of the main reasons that CONIKS was not adopted by any major corporations.

V. Conclusion

Throughout this paper, we have outlined the goals and objectives of the CONIKS infrastructure, as well as its difficulties. CONIKS strived to be an end-user key verification service built on the fundamental concept of key transparency. This is an extremely attractive selling point for a key verification service, as end-users benefit from the full transparency given by a decentralized service. However, this comes with many struggles and is not a trivial task. This was clearly shown when after CONIKS was presented at the USENIX conference in late 2015, it was not adopted by any companies. This was after a thorough review by engineers at Google, Yahoo, Apple, and Signal.

The main problem of CONIKS, as described in this paper, is its response to attacks and relatedly the key change protocol. The default key change protocol works by not requiring that a key change is signed by the private key of a user. While this increases ease of use for the user in some cases, such as key recovery, where the user must recover their private key, it struggles with false positives of malicious activity. This is due to the fact that when using the default key change policy, there are times when there is no proof of whether or not a key recovery was malicious, meaning that other clients do not know whether or not they can trust the user who attempted to recover their account. This was seen as one of the main barriers to CONIKS being adopted. The strict key change protocol was more secure but lacked usability. This is because it requires a user's private key for signing key changes, which means if the user's private key was ever lost, then the user has absolutely no way to recover it. Also if the user's key is ever stolen, then the adversary would be able to pose as the old user with the user unable to do anything about this. These were critiques of CONIKS listed by Marcela Melara in her 2016 article.
While CONIKS was never adopted despite being advertised to many large messaging identity providers, it was attempting to overcome a very difficult problem that remains unsolved in today's world. Many of these companies, in our opinion, may not have adopted CONIKS for both the reasons listed above as well as the fact that they have little incentive to adopt a decentralized key authentication service. This is because we do not believe that a large percentage of end-users are either aware of the key transparency that would be used if CONIKS was adopted, and if they are aware, they may not share the opinion that a decentralized key authentication service is necessarily better than a centralized one. There do exist end-users who care greatly about the security of their messages and the authentication protocol that is used in their software, but they are few and far between. The entire point of CONIKS is that the users do not have to unilaterally trust a potentially malicious identity provider, which is a very valid concern and a problem that still exists.

However, although CONIKS itself was never adopted, the developers still remain hopeful that a version of it will eventually be deployed sometime in the future. They stated that although overcoming the remaining barriers in CONIKS' development is not a minor endeavor, they remain optimistic that transparent key management is within their reach.

References

[1] P. Everton, "Google's gmail hacked this weekend? tips to beef up your security," 2013.

[2] N. Perloth, "Yahoo breach extends beyond yahoo to gmail, hotmail, aol users," *New York Times Bits Blog*, 2012.

[3] D. Kales, O. Omolola, and S. Ramacher, "Revisiting user privacy for certificate transparency," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 432–447, IEEE, 2019.

[4] R. C. Merkle, "A certified digital signature," in *Advances in Cryptology — CRYPTO' 89 Proceedings* (G. Brassard, ed.), (New York, NY), pp. 218–238, Springer New York, 1990.

[5] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer, "Rfc2440: Openpgp message format," 1998.

[6] P. R. Zimmermann, *The official PGP user's guide*. MIT press, 1995.

[7] L. Franchesci-Bicchierai, "Even the inventor of pgp doesn't use pgp," Sep 2015.

[8] A. Katwala, “We’re calling it: Pgp is dead,” May 2018.
[9] T. Fox-Brewster, “Whatsapp adds end-to-end encryption using textsecure,” Nov 2014.
[10] B. Schneier, “Apple’s imessage encryption seems to be pretty good,” 2015.
[11] G. B. Spendlove, *Security Analysis and Recommendations for CONIKS as a PKI Solution for Mobile Apps*. Brigham Young University, 2018.
[12] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, “Coniks: Bringing key transparency to end users,” 2014. https://eprint.iacr.org/2014/1004.pdf.
[13] M. Melara, “Why making johnny’s key management transparent is so challenging,” *Freedom To Tinker*, 2016.