Building a better YubiKey

Christopher Patton

University of Florida

Abstract. The aim of *two-factor authentication* is to tie a user's identity to something they *know* (their password) and something they *have* so that the attacker requires both in order to impersonate them. The latter is often realized using a "hardware token," a physical device that interacts with the server to authenticate the user. *YubiKey*, the flagship product of Yubico, is a low-cost, USB-enabled hardware token capable of supporting a variety of industry standard protocols. Each token also supports *Yubico OTP* ("one-time password"), a simple protocol based on symmetric-key cryptography that is designed to work out-of-the-box without the need to install code on the client. But in terms of usability, this ease-of-deployment comes at a significant cost: we find that a YubiKey can only be used to authenticate to one service securely. This work considers means by which a YubiKey may be used securely with many services, while keeping deployment complexity at a minimum. Going further, we propose a redesign of the token that would make the hardware platform useful for a wide variety of applications beyond OTP-based authentication.

Keywords: two-factor authentication, hardware token, YubiKey

1 Introduction

A crucial point of failure in virtually all web services is the user's password. Passwords that are easy to remember are often easy to guess, a reality which universities, companies, and governments alike must cope with. Today, token-based, two-factor authentication is, arguably, the most viable way to improve security of users' credentials. A number of companies manufacture *hardware tokens* for this purpose. One of the leaders in this industry is Yubico, whose flagship product, the *YubiKey*, supports a variety of industry standard protocols, including OATH HOTP [MBH^{+}05] and U2F [FID].

YubiKeys also provide authentication out-of-the-box via the *Yubico OTP* ("one-time-password") protocol. It involves the *YubiCloud*, a web service provided by Yubico. When the user plugs her token into a computer's USB port and presses a button, the token types out a string just as a keybaord would. The 44-character string, called a *one-time password* (OTP), looks something like this:

```
ccccccflivlifcdgrtgkhcjdfjcbljlhvehufurhtjlg
```

This string is typically entered into a form on a web page; when received, the web server sends the OTP to YubiCloud for verification. The first 12 characters denote the tokens's *public serial identifier*. The remaining 32 characters encode a 128-bit string, which is the AES encryption (under a key K stored on the token) of, among other things, the token's *private serial identifier* and a *counter*, which gets incremented each time the button is pushed. YubiCloud verifies the authenticity of the OTP by decrypting it and checking (1) that the private serial identifier matches its records and (2) that the OTP is fresh, meaning the counter is larger than the last authentic OTP it received. YubiCloud looks up the decryption key K using the public serial identifier that was transmitted in the clear.

Yubico OTP has a few key advantages over competing protocols. First, it works without installing any software on the client's system. The token presents itself as a USB keyboard and hence can be used with every major operating system without additional configuration. At the same time, it provides much better security than competing symmetric-key authentication protocols, such as the popular OATH TOTP protocol (e.g. Google Authenticator), which involves much shorter OTPs. Protocols based on public-key cryptography, such as U2F, inherently provide better security than their symmetric-key counterparts because they do not involve shared secrets. Nevertheless, there are a number of reasons to favor symmetric-key techniques, and Yubico OTP in particular. First of all, U2F requires client-side support, and as of this writing, only Opera, Firefox, and Chrome support its. Another reason to favor symmetric key solutions is that the hardware requirements are much smaller.

The problem of having a shared secret may be partially overcome by using a *hardware security module* (HSM) to manage the state associated with each token. The state is encrypted using a key that (ideally) never leaves the HSM boundary. To validate an OTP, it and the encrypted state are given to the HSM, which decrypts the state, validates the OTP, then updates the state. Yubico manufactures a low-cost, market-grade HSM, called *YubiHSM* for this purpose. It can be used to provide an authentication service similar to YubiCloud.

Security of Yubico OTP. This work aims to clarify the security that Yubico OTP provides and doesn't provide. Towards a formal treatment, we consider its security in the adversarial model of Bellare-Rogaway for entity authentication [BR94]. In this model, the adversary is assumed to utterly control the network and so is responsible for delivering messages between players in the protocol. It may inject, delay, reorder, replay, and drop messages at will. The players are the set of *clients*, the set of *servers* requesting authentication of the clients, and the *authority* that verifies OTPs (e.g., YubiCloud). Perhaps unintuitively, we also model the set of *tokens* (i.e. the YubiKeys) managed by the authority as players in the protocol. (Hence, the adversary is also responsible for "delivering" the OTP to the client.) This allows us to capture the idea that the protocol is "initiated" by the client physically interacting with the token.

We find no inherent flaws in the protocol when each YubiKey is used to authenticate to *at most one server*. In fact, we prove under standard assumptions about AES that the probability that an adversary is able to forge an OTP to the authority is roughly birthday bounded. However, if a YubiKey is used with more than one server, than Yubico OTP *does not suffice for security in the Bellare-Rogaway model*. In particular, an OTP intended for authenticating the client to one server can be intercepted by the adversary and rerouted to another service that accepts the same token as a credential.

OTP1. On the positive side, we suggest a simple modification to the protocol that offers a defense against rerouting attacks. In our protocol, the private identity of the token is replaced with the identity of the service requesting authentication, allowing us to cryptographically bind the service to the OTP. The protocol requires no modification to the token itself, and adds no additional overhead (computational, communication, or bandwidth). However, it does require a small amount of client-side code. This is unavoidable, since in order to bind the OTP to the service, it is necessary to send the token a message. First the client's system must be capable of talking to smart cards. (We find that both Ubuntu 16.04 and Mac OS Sierra do so without any additional configuration; we didn't test on Windows.) Second, it requires the installation of a small amount of code.

OTP2. We also take the liberty of rethinking the design of the YubiKey altogether. Ours results so far point to a fundamental axiom for token-based authentication: either the OTP or the token itself must be bound to the service requesting authentication. The latter is preferable, since it allows us to use a single token for many services. In practice, this means that the token will require some client-side support. While we're at it, why restrict ourselves to OTPs? We propose a modified token that supports a number *modes of operation*, including simple OTP-based authentication, request-bounded OTP-based authentication (the implicit goal of OTP1), message authentication, and encryption. We also consider a few concrete applications. For example, we show a way to provide two-factor authentication secure against dictionary attacks on the client's password with the same round complexity as the simple OTP protocol and with very little computational overhead. We also suggest an extension to TLS that provides client authentication via hardware tokens. Our proposal will require modifying the YubiKey firmware at a minimum. We suspect, however, that it can be implemented without modifying the hardware.

Organization of this report. Section 2 presents related work. Section 3 describes the tokens (YubiKey and YubiHSM) and their functionalities. Section 4 describes the OTP protocol and its security. Section 5 presents OTP1, and section 6 presents OTP2 and applications.

2 Background and prior work on YubiKeys

Despite their many pitfalls, passwords have long been understood as the most practical trade-off between security and usability for authentication. Early adopters recognized the need to store passwords securely [MT79], yet no manner of storage is adequate if the password can easily be guessed. This has lead system admnistrators to enforce policies for selecting passwords (for example, adding special characters, or requiring a minimum length) and periodically changing them. In their landmark user study, Adams and Sasse [AS99] point out that, paradoxically, these policies *adversely* affect security, because they drive users to choose weak passwords, or even to write them down. In light of this finding, the usability of any technique to enhance password-based authentication has become a first-class concern.

A number of approaches to supplement (or supplant) passwords have been proposed, each having unique barriers to adoption. Bonneau et al. [BHOS12] point out that *deployability* of the mechanism is a critical factor. In recent years, two-factor authentication (2FA) has emerged as an approach with a reasonable trade-off between security, usability, and deployability. Typically the second factor is realized using something in possession of the user, such as a cell phone or a dedicated hardware token. The simplest protocol works as follows. The authority associates a unique secret to each client. When the client requests a log in, the authority uses the secret and current time to generate a short sequence (6-8) of digits, which it sends to the client's phone. Both this sequence and the password are needed to log in. Each "one-time password" (OTP) is valid only for a short time (typically 30 seconds). While this system is easy to deploy, it suffers from a major security flaw; the OTP is transmitted via SMS, an insecure channel with a number of known vulnerabilities [RST$^+$16]. (For example, vulnerabilities in SS7 [Eng08] have been exploited to intercept OTPs in 2FA systems used for bank accounts [TZ17].)

An improvement came with a pair of protocols by OATH, the Initiative for Open Authentication. The first of these is called TOTP [MMPR11] and works as above, except the client's phone (or token) shares a key with the server. This change affectively removes the SMS attack vector. TOTP enjoys wide adoption; a prominent example is the Google Authenticator app for Android, iOS and other platforms [Goo]. The second, HOTP [MBH$^+$05], is a variant of TOTP that uses a counter instead of a timestamp. An early hardware token employing a protocol similar to TOTP was RSA's SecurID [RSA]. Yubico OTP is similar to HOTP, in that it generates one-time passwords using a stateful counter; however, it has several advantages from a security standpoint. (For one, Yubico OTPs are much longer, and so harder to forge.)

Each of these protocols, including Yubico OTP, has an important drawback; they require that the authenticating party keeps a secret. Indeed, in 2011, RSA SecureID's server was breached, leading to the exposure of a number of tokens' secrets [Kam11]. This lead Yubico to develop a low-cost, hardware-security module (the YubiHSM) in order to simplify the deployment of a secure authentication server. Of courese, public-key cryptography affords the opportunity to avoid this drawback altogether. The FIDO Allience, in collaboration with a number of industry leaders, has developed the universal two-factor (U2F) protocol [FID], which usess a private key stored on a hardware token; the corresponding public key is stored on the server. YubiKeys support this more sophisticated protocol in addition to Yubico OTP. The latter has the distinct advantage of being easier to deploy; U2F requires the web browser to support it, since it involves a couple rounds of communications between the token and the server. Yubico OTP requires no browser support.

Due to its simplicity, Yubico OTP remains an important industry player. As such, it has received a respectable amount of attention in academia. The first and only formal treatment of the protocol was provided by Künnemann and Steel [KS13], but their result is limited in a fundamental way. They work in a restricted adversarial model, called the *Dolev-Yao* model [Her05]. Roughly speaking, they prove that no Dolev-Yao adversary seeing an unbounded number of OTPs can recover the underlying secret. This is a weaker security property than we generally hope for. In particular, the stronger *Bellare-Rogaway* model [BR94] directly captures an adversary's ability to forge the users's credentials. From this perspective, the exact security of Yubico OTP remains open.

Künneman and Steel also pointed out attacks on the YubiHSM that, with a particular configuration, allows an attacker on the authority's system to decrypt sate associated with YubiKeys. YubiKey has also been shown to be susceptible to side-channel attacks. Oswald et al. [ORP13] use minimally-invasive power analysis to recover the token's secret key for generating OTPs.

3 YubiKey and YubiHSM

This section provides an overview of the functionalities exposed by the hardware tokens, namely the YubiKey (denoted KEY) and the YubiHSM (denoted HSM). Before we begin, we note that the protocol involves three principals: the *client* being authenticated, the *server* requesting authentication of the client, and the *authority* that authenticates the client to the server. The client possesses KEYand the authority possesses HSM.

YubiKey 4. There are four varieties of tokens: two are have a large profile designed to be carried by the client, and two have a small profile designed to remain in the client's machine. Yubico manufactures a USB-A and USB-C variant for each profile type. All four varieties support the same suite of protocols.

The token has two (re)programmable slots. A slot is configured to run one of a variety of protocols, including Yubico OTP, OATH-HOTP, challenge-response (either OTP- or HMAC-based), or it can be programmed to output a static password. The protocol is activated by pressing (and holding) on the key. The first slot is activated by pressing the button and immediately letting go; the second slot is activated by pressing and holding for a few seconds. All of the values stored on the token, including the shared secret, are generated by the programmer and written to the device. The programmer can choose to disable configuration; from that point on, security-critical values cannot be overwritten. (3.1: Chris says: Can a slot configured for, say, OTP-based CR, be used in a different way, say HMAC-based CR?)

The token presents three interfaces to the operating system: the first two are *human interface devices* (HIDs) and the third is a *smart card*. The two HIDs correspond to the two slots and are used for the non-interactive protocols, in particular Yubico OTP, OATH-HOTP, and static password. The challenge-response protocols require the client to send a message to the device; this is handled by the smart card interface. (3.2: Chris to-do: Double check with Dave.)

In this work we are interested in OTP and OTP-based challenge-response. We write execution of the OTP protocol as $(otp, \sigma') \leftarrow \text{KEY}(\text{“otp”}, \sigma)$, where otp denotes the OTP, σ denotes the key's internal state, and σ' denotes the token's update internal state. (We emphasize that the client gets the OTP, but not the token's state.) Execution of OTP challenge-response is written $(Y, \sigma') \leftarrow \text{KEY}(\text{“chal”}, m, \sigma)$, where m is a 6-byte string called the *challenge*, and Y is a 16-byte string called the *response*.

YubiHSM 1.6. The HSM is designed to provide secure management of shared secrets in protocols involving YubiKeys, but it also offers a few other cryptographic functionalities that make it useful for other applications. A properly configured YubiHSM can be used to improve security. The client needn't trust the authority to keep a secret; she need only trust that the HSM is properly configured. (3.3: Chris says: The configuration of the YubiHSM constitutes an intricate attack surface, one that warrants further study.) All operations are based on symmetric cryptography. (However, the newer YubiHSM 2, described below, also supports a few public-key operations.) The token provides the operating system with two serial interfaces: one for configuration, and the other for executing protocols. (3.4: Chris to-do: Double check this with Dave.)

The token is configured to operate in one of two modes: *HSM* and *WSAPI*. The latter facilitates easy deployment of Yubico OTP for a limited number of keys, while the latter offers more features. Changing the mode of operation requires resetting the token, and resetting requires physical access. (It also requires physical access to configure the device.)

HSM. This mode offers a number of features, including OTP validation, management of shared secrets, and a handful of generic operations. Each operation is associated with a *key handle* for a key generated by and stored within the token. (The HSM holds up to 40 keys.) Also associated to each key is a set of *permitted operations* using that key. All keys 128-bit strings generated using the HSM's random number generator. (3.5: Chris says: The key store itself is "encrypted using AES-256" in non-volatile memory. Where is does the key come from for encrypting the key store?") Each key can be configured to perform the following operations:

- *Key management.* The HSM can be used to wrap keys for external storage. Keys can then be unwrapped within the HSM to be used as a *temporary key* (accessed with special handle). AES-CCM is used for key (un)wrapping. (3.6: Chris says: It would be interesting to see how Tom's crypto API paper [SSW16] applies to YubiHSM key wrapping.)
- *External Yubico OTP validation.* The HSM stores the key, public and private identifiers, and counter state associated with each YubiKey it manages. The state is stored externally, using AES-CCM is used for confidentiality and integrity.
- *Internal Yubico OTP validation.* The HSM can store the state of up 1024 YubiKeys in its internal database.
- *Message authentication.* A key handle can be used to authenciate a message using HMAC-SHA1.
- *"Plain" encryption/decryption.* A key may be used in AES-ECB mode. (3.7: Chris says: This amounts to a blockcipher oracle. If a key that is used for AES-CCM is also used for AES-ECB mode, then ciphertext can easily be decrypted using an attack by [KS13]. The HSM is still vulnerable to this attack, since a key may be configured to work in both modes. I wonder what other "bad" configurations there are.)

Another feature of the HSM is *pseudorandom number generation*. Entropy is gathered from timing difference between input/output events. The PRNG can also be reseeded using entropy provided by the user. The PRNG is also used to generate keys stored on the HSM. (3.8: Chris says: Dave is interested in chocking the entropy of the PRNG. I'm wondering what algorithm is used.)

WSAPI. The operations above can be used to implement the Yubico OTP protocol as described in Section 4 without exposing any sensitive values to the authority. The WSAPI mode[1] allows the protocol to be implemented conveniently for a limited number of YubiKeys. This is the "mainstream" mode of operation for the HSM, as it supports the Yubico OTP protocol with a minimal attack surface. A major limitation, however, is that this mode stores all secrets internally; thus, it is only useful for a limited deployment.

YubiHSM 2. Recently, Yubico released its new version of the YubiHSM, which offers a number of new features. It supports the common crypto API standards Microsoft CNG and PKCS#11, in addition to the native API above. (3.9: Chris says: We should check that the API is actually the same.) It also supports new crypto operations, such SHA2, RSA- and EC-based signing/encryption, and attestation of the HSM's state and configuration.

4 The Yubico OTP protocol

This section specifies the Yubico OTP protocol in the manner in which it is generally deployed.[2] In order to simplify the exposition, we will not assume the use of a YubiHSM; note, however, that the HSM could be used to keep all sensitive values hidden from the authority, effectively reducing trust in the authority to trusting that the HSM has been properly configured. We begin with a bit of notation.

[1] See https://developers.yubico.com/yubikey-val/Validation_Protocol_V2.0.html.

[2] See https://developers.yubico.com/OTP/Specifications.

Notation. If n is a positive integer, let $[1..n]$ denote the set of integers from 1 to n. Variables are strings unless noted otherwise. Strings are finite sequences of bytes, i.e., elements of $(\{0,1\}^8)^*$, unless noted otherwise. Let ε denote the empty string. Let X and Y be strings and let $X \,\|\, Y$ denote their concatenation. String indexing has the same semantics as the Python programming language: let $0 \le i \le j \le |X|-1$. Then $X[i]$ denotes $(i+1)$-th byte of X and $X[i{:}j]$ denotes the sub string $X[i] \,\|\, \cdots \,\|\, X[j-1]$ of X. Let $X[i{:}] = X[i{:}|X|-1]$ and $X[{:}j] = X[0{:}j]$. If $i \ge j$, then $X[i{:}j] = \varepsilon$ by convention. Let $\langle X \rangle$ denote the modhex encoding of X.[3] Let $[i] \in \{0,1\}^8$ denote the byte encoding of integer $i \in [0..255]$.

The data frame. The KEYstores a 16-byte key K. When activated, it emits a 44-byte string, called a *one-time password*. An OTP is a modhex string $\langle I \,\|\, Y \rangle$, where $|I| = 6$ and $|Y| = 16$. String I is the *public identifier* of KEYand $Y = E_K(X)$, where E denotes the AES-128 block cipher and X is a 16-byte string called the *data frame*. The data frame encodes the following quantities:

- string m ($X[{:}6]$) — the payload.
- integer *session_ctr* ($X[6{:}8]$) — the session counter. Changes each time *token_ctr* rolls over.
- integer *time_low* ($X[8{:}10]$) — the low-order time stamp bits. Changes quickly.
- integer *time_high* ($X[10]$) — the high-order time stamp bits. Changes slowly.
- integer *token_ctr* ($X[11]$) — the token counter. Changes each time KEYis invoked.
- string r ($X[12{:}14]$) — a pseudorandom value generated by KEY. (4.1: Chris says: How is this seeded? Can this be reseeded?)
- integer *crc* ($X[14{:}]$) — a checksum of $X[{:}14]$, denoted SUM($X[{:}14]$).[4]

The payload is the *private identifier* of KEYin the Yubico OTP protocol. An OTP is deemed *authentic* if the payload of the decrypted frame matches the private ID the authority has on record *and* the checksum is correct. An OTP is *fresh* if the counter is larger than the previous frame. To enable this functionality, it suffices to have the authority store the frame of the last authentic and fresh OTP it received. The *initial frame* is defined by setting the payload to the private ID, setting the counters, time stamp, and pseudorandom bytes all to zero, and computing the checksum of the frame. Define $X.ct$ as $X.token\_ctr + (X.session\_ctr \ll 8)$, where the $\ll$ dentos the bit shift operator.

The protocol. Let $\mathcal{C}$ denote the client in possession of KEY, $\mathcal{S}$ denote the server, and $\mathcal{A}$ denote the authority. The server and authority share a key K' for HMAC-SHA1, denoted MAC. The protocol works as follows:

1. $\mathcal{C}$ executes $(otp, \sigma') \leftarrow \textsc{key}(\text{“otp”}, \sigma)$ and transmits otp to $\mathcal{S}$.
2. $\mathcal{S}$ computes $H = \textsc{mac}_{K'}(J \,\|\, N \,\|\, otp)$ and sends (H, J, N, otp) to $\mathcal{A}$, where N is a nonce and J is $\mathcal{S}$'s identity.
3. $\mathcal{A}$ looks up the key K' associated with J and checks that $H = \textsc{mac}_{K'}(J \,\|\, N \,\|\, otp)$. If so, it proceeds to step 4; otherwise it halts.
4. $\mathcal{A}$ decodes $\langle I \,\|\, Y \rangle \leftarrow otp$ and looks up the key K and data frame T associated with I. It computes $X \leftarrow E_K^{-1}(Y)$, checks that otp is authentic (that $X.m = T.m$ and $\textsc{sum}(X[{:}14]) = X.crc$) and fresh (that $X.ct > T.ct$). If both conditions hold, it lets $T \leftarrow X$. It lets R denote the result of check. Finally, it computes $H' = \textsc{mac}_{K'}(N \,\|\, R \,\|\, otp)$, and sends (H', N, R, otp) to $\mathcal{S}$.
5. $\mathcal{S}$ checks that $H' = \textsc{mac}_{K'}(N \,\|\, R \,\|\, otp)$ and that the response R says that otp is authentic and fresh. If so, it accepts; otherwise it rejects.

Initialization. The key K and initial frame T are generated and used to compute the KEY's initial state σ, and K and T are given to $\mathcal{A}$. Next, key K' is generated and given to $\mathcal{A}$ and $\mathcal{S}$. The public identity I of KEY is given to $\mathcal{S}$, and the identity J of $\mathcal{S}$ is given to $\mathcal{A}$.

Verification of H is optional. Computing the MAC of the server request in step 2 and verification of MAC in step 3 is an optional feature of the protocol. This step is not essential for authenticating the client, but may be useful for other purposes.

Interpreting the result R. The contents of R are specified by the protocol.[5] If the otp is valid, then it includes the counter and the time stamp. Otherwise it indicates what went wrong; for example, the otp was mal-formed, inauthentic, or not fresh.

[3] OTPs use the alphabet `cbdefghijklnrtuv` instead of the usual `0123456789abcdef` for encoding nibbles.

[4] Computed as `0xffff` minus the output of `https://github.com/Yubico/yubico-c/blob/master/ykcrc.c`.

[5] See `https://developers.yubico.com/OTP/Specifications/OTP_decryption_protocol.html`.

```
$\mathbf{Exp}^{\text{forge}}_{E,S}(\mathcal{F})$
$\mathcal{E} \leftarrow \emptyset$; $K \twoheadleftarrow \mathcal{K}$; $T \leftarrow S$; $win \leftarrow 0$; $\mathcal{F}^{\textbf{Key},\textbf{Auth}}(S)$
return $win$

$\mathbf{Key}(\,)$
$S.ct \leftarrow S.ct + 1$; $Y \leftarrow E_K(S)$; $\mathcal{E} \leftarrow \mathcal{E} \cup \{Y\}$; return $Y$

$\mathbf{Auth}(Y)$
$X \leftarrow E_K^{-1}(Y)$
if $\textsc{isValid}(X, T)$ then
    if $Y \notin \mathcal{E}$ then $win \leftarrow 1$
    $T \leftarrow X$
return $X$
```

```
$\mathbf{Exp}^{\text{sprp}}_{E,b}(\mathcal{D})$
$K \twoheadleftarrow \mathcal{K}$; $\pi \twoheadleftarrow \mathrm{Perm}(n)$
if $b = 1$ then $b' \twoheadleftarrow \mathcal{D}^{E_K(\cdot), E_K^{-1}(\cdot)}$
else $b' \twoheadleftarrow \mathcal{D}^{\pi(\cdot), \pi^{-1}(\cdot)}$
return $b'$
```

```
$\mathbf{G}^{c}_{S}(\mathcal{F})$
$\mathcal{E} \leftarrow \emptyset$; $T \leftarrow S$; $win \leftarrow 0$; $\mathcal{F}^{\textbf{Key},\textbf{Auth}}(S)$
return $win$

$\mathbf{Key}(\,)$
$S.ct \leftarrow S.ct + 1$
if $S \notin \mathrm{Dom}\,\pi$ then $\pi(S) \twoheadleftarrow \{0,1\}^n \setminus \mathrm{Rng}\,\pi$
$\mathcal{E} \leftarrow \mathcal{E} \cup \{\pi(S)\}$; return $\pi(S)$

$\mathbf{Auth}(Y)$
if $Y \in \mathcal{E}$ then $X \leftarrow \pi^{-1}(Y)$
    if $\textsc{isValid}(X, T)$ then $T \leftarrow X$
    return $X$
$X \twoheadleftarrow \{0,1\}^n$
if $X \in \mathrm{Dom}\,\pi$ then
    bad $\leftarrow 1$; if $c = 1$ then $X \twoheadleftarrow \{0,1\}^n \setminus \mathrm{Dom}\,\pi$
$\pi^{-1}(Y) \leftarrow X$
if $\textsc{isValid}(X, T)$ then
    $win \leftarrow 1$; $T \leftarrow X$
return $X$
```

Fig. 1: Let $E : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a blockcipher, $\mathcal{K}$ be a finite set, and $n \in \mathbb{N}$. Let $\mathrm{Perm}(n)$ denote the set of all permutations over $\{0,1\}^n$. Let $\textsc{isValid}(X, T)$ denote the predicate $(X.m = T.m) \wedge (X.crc = \textsc{sum}(X[:14])) \wedge (X.ct > T.ct)$. **Top-left:** the FORGE game for E; **Bottom-left:** the SPRP game for E; and **Right:** a game used to fix Lemma 1.

4.1 Security

We describe, informally, the goal of the Yubico OTP protocol. Fix $c, s, t \in \mathbb{N}$. The players are the set of clients $\{\mathcal{C}_i\}_{i \in [c]}$, the set of servers $\{\mathcal{S}_j\}_{j \in [s]}$, the set of tokens $\{\textsc{key}_k\}_{k \in [t]}$, and the authority $\mathcal{A}$. A client may be in possession of any number of tokens, but no token is possessed by more than one client. Clients "register" at most one token with each server, meaning the server is given the public identity of the token. A client may register any token with any server, and a single token may be used for any number of servers.

Threat model. We adopt the threat model of [BR94]. The adversary is active and controls all flows between players. We assume that the players are initialized (as described above) before the attack begins. Instances of the protocol, called *sessions*, may be carried out simultaneously. Each session is initialized by the adversary and is defined by a triple of positive integers (i, j, k). A *valid session* is one in which the Yubico OTP protocol involving $\mathcal{C}_i$, $\mathcal{S}_j$, $\textsc{key}_k$, and $\mathcal{A}$ is carried out faithfully, and where $\mathcal{C}_i$ is "in possession of" $\textsc{key}_k$. A session is not valid unless it completes. A session is called *accepting* if $\mathcal{S}_j$ accepts in that session. The goal of the adversary is to get *any* server to accept in an *invalid* session. We say the protocol is secure if the probability that any "reasonable" adversary can do so is negligible.

Physical assumptions. In our adversarial model, we assume that the tokens expose the interface described in Section 3, and otherwise offer no attack surface. Of course, this is a strong assumption that does not stand up to scrutiny. For example, prior work has shown YubiKeys to be susceptible to side-channel attacks [ORP13]. Thus, an adversary in control of a client's system, or in physical proximity of the token, may have access to additional information not captured in the model. (4.2: Chris says: This might make an interesting attack surface, but I'll ignore it for the time being.)

In the following, we consider the various avenues of attack available to our adversary.

Forging the authority response. The use of HMAC-SHA1 and the nonce N in the server response ensure that the response is authentic and cannot be replayed. (Each server must take care to make sure that it never repeats a nonce.) The MAC also binds the response to the OTP, ensuring that if the OTP is valid, then the response is valid. In the remainder, we focus on the validity of the OTP.

Forging an OTP. A successful forgery is an attack whereby the adversary transmits an OTP to a server that the authority deems authentic and fresh, but was not output by the token. An OTP is merely the AES encryption of some "stuff" using a key shared between each $\textsc{key}_k$ and $\mathcal{A}$, and so it is natural to reduce an adversary's ability to forge to the security of AES. Typically we would assume that AES is a good *pseudorandom permutation*, meaning that no "reasonable" adversary can distinguish the output of the blockcipher from a true random permutation, even if the adversary chooses the inputs. However, this does not suffice for our setting, since the forgery adversary has, in effect, an oracle for the permutation *and* its inverse. A more appropriate assumption is that AES a is a *strong pseudorandom permutation*.

Let $n \in \mathbb{N}$, $\mathcal{K}$ be a finite set, and let $E : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a blockcipher. We associated to E, an adversary $\mathcal{D}$, and bit b a game, SPRP, defined in Figure 1. If $b = 1$, then a key K is chosen uniformly from the set $\mathcal{K}$ and $\mathcal{D}$ is given access to oracles $E_K(\cdot)$ and $E_K^{-1}(\cdot)$. Otherwise, a permutation π is chosen uniformly from the set of all permutations over $\{0,1\}^n$ and the adversary is given oracles for π and its inverse. Eventually it outputs a bit b', which is returned by the game. We define the advantage of $\mathcal{D}$ in attacking E as

$$\mathbf{Adv}_E^{\text{sprp}}(\mathcal{D}) = \left|\Pr\left[\,\mathbf{Exp}_{E,1}^{\text{sprp}}(\mathcal{D}) = 1\,\right] - \Pr\left[\,\mathbf{Exp}_{E,0}^{\text{sprp}}(\mathcal{D}) = 1\,\right]\right| .$$

Informally, we call E a strong pseudorandom permutation if $\mathbf{Adv}_E^{\text{sprp}}(\mathcal{D})$ is “small” for all “reasonable” $\mathcal{D}$.

We note that there is a significant gap between PRP and SPRP security. On the theoretical side, there are constructions of blockciphers that are provably secure as PRPs, but are *not* SPRPs, e.g. the Feistel constructions of [LR88]. On the other hand, the SPRP security of AES has undergone significant cryptanalysis, e.g. using the boomerang attack of Wanger [Wag99]. It is generally believed (and often assumed) that AES-128 is a secure SPRP.

One client, one token, and one server. Under the SPRP assumption, we can prove a useful lemma that helps us quantify the adversary’s ability to forge an OTP. Suppose for the moment that there is just one client with a single token registered with one server. We may formalize the forgery attack as the game FORGE defined in Figure 1. The game is parameterized by an *initial data frame* S, which specifies the shared state between the token and authority before the attack begins. The adversary $\mathcal{F}$ is given an oracle **Key** for the token and an oracle **Auth** for authenticating OTPs. The game sets a flag win if the adversary makes an **Auth**-query that is fresh and authentic, but was never output by **Key**. The outcome of the game is the value of win when $\mathcal{F}$ halts. We define the advantage of $\mathcal{F}$ in forging against E (beginning at state S) as

$$\mathbf{Adv}_{E,S}^{\text{forge}}(\mathcal{F}) = \Pr\left[\,\mathbf{Exp}_{E,S}^{\text{forge}}(\mathcal{F}) = 1\,\right] .$$

Lemma 1. *Let $n = 128$, $\mathcal{K}$ be a finite set, and $E : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a blockcipher. Let $S \in \{0,1\}^n$ be a data frame and let $\tau = S.ct < 2^{24}$. Let $\mathcal{F}$ be a* FORGE *adversary making $\kappa < 2^{24}$ queries to **Key** and α queries to **Auth**. There exists an* SPRP *adversary $\mathcal{D}$ such that*

$$\boldsymbol{Adv}_{E,S}^{\text{forge}}(\mathcal{F}) \le \boldsymbol{Adv}_E^{\text{sprp}}(\mathcal{D}) + \frac{\alpha^2 + 2\kappa\alpha}{2^{129}} + \frac{\alpha(2^{24} - \tau)}{2^{88}}$$

where $\mathcal{D}$ has the same runtime as $\mathcal{F}$.

Proof. Adversary $\mathcal{D}$ is constructed by simulating $\mathcal{F}$ in its game in the natural way. In particular, calls to the blockcipher (i.e. the call to E in **Key**) are forwarded to $\mathcal{D}$’s first oracle, and calls to the inverse of the blockcipher (i.e. the call to E^{-1} in **Auth**) are forwarded to $\mathcal{D}$’s second oracle. Once $\mathcal{F}$ halts, adversary $\mathcal{D}$ halts and outputs 1 if $(\mathit{win} = 1)$ and 0 otherwise.

Let b denote the challenge bit in $\mathcal{D}$’s game. If $b = 1$, then the simulation is perfect; otherwise $\mathcal{D}$’s output is identically distributed to the output of $\mathbf{G}_S^1(\mathcal{F})$, where game $\mathbf{G}^1$ is as defined in Figure 1. (This game is the same as the FORGE game instantiated with a random permutation rather than a blockcipher. The permutation is simulated via lazy evaluation.) It follows that

$$\begin{aligned}\mathbf{Adv}_E^{\text{sprp}}(\mathcal{D}) &= \Pr\left[\,\mathbf{Exp}_{E,1}^{\text{sprp}}(\mathcal{D}) = 1\,\right] - \Pr\left[\,\mathbf{Exp}_{E,0}^{\text{sprp}}(\mathcal{D}) = 1\,\right] \\ &= \Pr\left[\,\mathbf{Exp}_{E,S}^{\text{forge}}(\mathcal{F}) = 1\,\right] - \Pr\left[\,\mathbf{G}_S^1(\mathcal{F}) = 1\,\right] .\end{aligned}$$

We next note the random variables $\mathbf{G}_S^1(\mathcal{F})$ and $\mathbf{G}_S^0(\mathcal{F})$ are identically distributed until either game sets the flag bad $\leftarrow 1$. This occurs if $\mathcal{F}$ asks $\mathbf{Auth}(Y)$, where Y was never output by **Key**, and X chosen uniformly from $\{0,1\}^n$ is in the domain of π. On the i-th query to **Auth** This occurs with probability at most $(\kappa + i - 1)/2^n$. Summing over all α queries yields

$$\left|\Pr\left[\,\mathbf{G}_S^1(\mathcal{F}) = 1\,\right] - \Pr\left[\,\mathbf{G}_S^0(\mathcal{F}) = 1\,\right]\right| \le \frac{2\kappa\alpha + \alpha^2}{2^{n+1}} .$$

We need only bound $\Pr\left[\,\mathbf{G}_S^0(\mathcal{F}) = 1\,\right]$. In this game, each query to $\mathbf{Auth}(Y)$ such that Y was never output by **Key** samples an X from $\{0,1\}^n$ independently of all previous queries. Hence, the probability that that win gets set, i.e. $X.m = T.m$, $X.\mathit{crc} = \textsc{sum}(X[:14])$, and $X.ct > T.ct$, is independent for each query. The payload $(X.m)$ is the first 48 bits (6 bytes), the checksum $(X.\mathit{crc})$ is the last 16 bits (2 bytes), and the counter $(X.ct)$ is comprised of 24 bits (3 bytes). Since these values are non-overlapping, there are at most $2^{n-88}(2^{24} - \tau)$ values for X that would set win. (The precise number depends on the number of OTPs "delivered" by the adversary, i.e. the number of queries $\mathbf{Auth}(Y)$ such that Y was output by **Key**.) Since X is a uniform random string, it follows that

$$\Pr\left[\, \mathbf{G}_S^0(\mathcal{F}) = 1 \,\right] \leq \frac{\alpha(2^{24} - \tau)}{2^{88}} \,.$$

This yields the final bound. □

Interpreting the bound. The Yubico OTP protocol achieves roughly birthday-bound (i.e., 64-bit, half the block length) security in the FORGE game, meaning the bound becomes vacuous after the adversary makes about $\alpha = 2^{64}$ forgery attempts. (Or somewhat less, depending on the value of κ, which is the number of OTPs the adversary gets to see.) The last term in the bound gets smaller as the initial counter value increases, but is at most $\alpha/2^{64}$ (i.e., when $\tau = 0$). Of course, this bound should not be interpreted as the concrete security for the overall protocol; it only captures one attack vector. However, we can conclude that the concrete security of the overall protocol is no better than this.

Rerouting an OTP. Lemma 1 captures an adversary's ability to forge an OTP when there is just one client, one token, and one server. More generally, suppose there are any number of clients, tokens, and servers, but each token is used to authenticate the client to at most one server. It's not difficult to see that security in this more general setting reduces to FORGE security. (Perhaps with a small cost in concrete security.) However, if we allow a client to register a single token with more than one server, then there is a trivial attack whereby the adversary simply *reroutes* an OTP meant for one server to another. Suppose that $\mathcal{C}_i$ has registered $\textsc{key}_k$ with servers $\mathcal{S}_j$ and $\mathcal{S}_{j^*}$. The adversary initiates a session (i, j, k) and runs it until $\textsc{key}_k$ outputs otp to $\mathcal{C}_i$. It then initiates a new session (i, j^*, k), but instead of faithfully running the protocol, it sends otp to $\mathcal{S}_{j^*}$, then runs the protocol from that point forward. In the end, $\mathcal{S}_{j^*}$ will accept, although the session is invalid.

Conclusion. The Yubico OTP protocol appears to provide strong (at least birthday-bound) security as long as a client never uses the same token for more than one service, since an adversary that can intercept an OTP can easily use it to authenticate to another service. A naïve solution would be to require the client and server to use TLS so that the OTP is not sent in the clear. But unless the client and server are mutually authenticated during the handshake, the channel is vulnerable to a man-in-the-middle attack. Since the OTP is itself a means of authentication, this is not a viable solution.

The private identifier is useless. The FORGE game is parameterized by the initial data frame, which models the shared state between the token and authority in the protocol. The data frame is given to the adversary in the game, Hence, the private identifier being secret is not essential for security. It suffices to have a public identifier and a secret key associated with it. As Lemma 1 indicates, the payload can be anything, for example, the all zero string. Indeed, many of the components of the data frame can be removed without compromising security: in particular, the checksum, the pseudorandom value, and time stamp. We only need the payload (some publicly-known value) and the counter.

5 OTP1

In this section, we consider means by which a YubiKey can be used securely for multiple services. The goal will be to bind the OTP to the service requesting authentication. We first recall that each YubiKey has two slots, which both can be configured to run Yubico OTP with different services. However, we will aim for a protocol that can be used with any number of services. Unfortunately, any such solution will involve *sending a message* to the token, which means we must forfeit a major advantage of Yubico OTP: the protocol requires minimal client-side support, since the client's system must only be capbble of recognizing a USB keybaord. Our solution will require the client to install a small amount of code,[6] and their system must be capable of talking to smart cards.[7]

OTP challenge-response. In Section 3 we briefly described the *challenge-response* mode of the YubiKey. In more detail, the server sends to the client a short (6-byte), random challenge. The client forwards this to their YubiKey, which transforms it into a data frame with the challenge as its payload. It outputs the encrypted data frame, which is forwarded to the server, then to the authority. The authority decrypts and checks that it is fresh and that the checksum bits match; if so, it sends the payload to the server. Finally, the server checks that the payload matches the challenge.

The challenge-response protocol above is sufficient to mitigate rerouting attacks as long as no two servers choose the same challenge. (Assuming these are chosen randomly, a collision is unlikely.) However, it adds an extra 0.5 round trip and requires the server to maintain a bit of extra state. We suggest the following alternative.

[6] See `https://github.com/Yubico/python-yubico`.

[7] Both Ubuntu 16.04 LTS and Mac OS 10.12.6 ("Sierra") are able to talk to the YubiKey without issue.

The revised protocol. The client *hashes the identity of the service*, and uses the first 6 bytes of the hash as the challenge. Let $\textsc{hash}$ be a cryptographic hash function whose output is at least 6 bytes in length. The new protocol is as follows:

1. Given J, the identity of $\mathcal{S}$, the client $\mathcal{C}$ computes $m \leftarrow \textsc{hash}(J)[6{:}]$ and executes $(Y, \sigma') \leftarrow \textsc{key}(\text{“chal”}, m, \sigma)$. It then transmits $otp = \langle I \,\|\, Y \rangle$ to $\mathcal{S}$. (5.1: Chris says: Not sure how to obtain I from the YubiKey.)
2. $\mathcal{S}$ computes $H = \textsc{mac}_{K'}(J \,\|\, N \,\|\, otp)$ and sends (H, J, N, otp) to $\mathcal{A}$, where N is a nonce and J is $\mathcal{S}$'s identity.
3. $\mathcal{A}$ looks up the key K' associated with J and checks that $H = \textsc{mac}_{K'}(J \,\|\, N \,\|\, otp)$. If so, it proceeds to step 4; otherwise it halts.
4. $\mathcal{A}$ decodes $\langle I \,\|\, Y \rangle \leftarrow otp$ and looks up the key K and data frame T associated with I. It computes $X \leftarrow E_K^{-1}(Y)$, checks that otp is authentic (that $X.m = \textsc{hash}(J)[6{:}]$ and $\textsc{sum}(X[{:}14]) = X.crc$) and fresh (that $X.ct > T.ct$). If both conditions hold, it lets $T \leftarrow X$. It lets R denote the result of check. Finally, it computes $H' = \textsc{mac}_{K'}(N \,\|\, R \,\|\, otp)$, and sends (H', N, R, otp) to $\mathcal{S}$.
5. $\mathcal{S}$ checks that $H' = \textsc{mac}_{K'}(N \,\|\, R \,\|\, otp)$ and that the response R says that otp is authentic and fresh. If so, it accepts; otherwise it rejects.

Initialization is the same as in the standard Yubico OTP protocol (Section 4), except that $\mathcal{C}$ is also given the identity J of $\mathcal{S}$.

5.1 Security

The changes to the protocol are minimal, and so we expect it to have the same security properties as Yubico OTP. In addition, it offers a defense against OTP rerouting. Suppose that J and J^* are the identity of two servers and $\textsc{hash}(J)[6{:}] \neq \textsc{hash}(J^*)[6{:}]$. Then an OTP generated in a session for (i, j, k) will not be deemed authentic in a session for (i, j^*, k).

This begs the question: how likely is it that the payloads for two server identities collide? We *cannot* bound this probability using the collision resistance of $\textsc{hash}$, since we need to truncate its output quite a bit in order to fit it in the frame. (For example, SHA1 outputs 20 bytes and SHA256 outputs 32, whereas the payload is just 6 bytes.) In the random-oracle model, we can argue that the probability is at most $s^2/2^{48}$, where s is the number of services. But even this bound is significantly weaker than what we would hope for. In particular, it is much weaker than the bound in Lemma 1.

Still, this change to the protocol is a viable option for allowing deployed YubiKeys to be used with multiple servers securely. But what if we're willing to modify the tokens? In the next section, we propose a way to significantly improve security. We suspect our proposal will require changing the firmware, but not the YubiKey hardware.

No YubiHSM support for this protocol. Note that the YubiHSM 1.6 does not support OTP challenge-response. Recall that in the Yubico OTP protocol, the expectation is that the payload remain private. But in the challenge-response protocol, it is necessary that the payload be sent to the server. In our protocol, the authority can check that the payload matches what is expected, but this does not appear to be supported. (5.2: Chris says: What about YubiHSM 2?)

6 OTP2

Co-opting the challenge-response mode as described above is a viable way to use YubiKeys with multiple servers, but there is certainly room for improvement. One way to improve security is to modify the frame data structure so that the payload is longer. The only essential component is the 3-byte counter, leaving 13 bytes for the payload. However, as long as we're modifying the YubiKey, we can do much better. In the following, we present a revised OTP protocol, called *OTP2*, that uses a *tweakable blockcipher* as a building block. Our design may be used in multiple *modes of operation*, supporting a wide variety of applications beyond OTP-based authentication.

A *tweakable blockcipher* [LRW11] is a deterministic algorithm $E : \mathcal{K} \times \mathcal{T} \times \{0,1\}^n \to \{0,1\}^n$, where n is a positive integer, $\mathcal{T}$ is a set, and $\mathcal{K}$ is a finite set such that for each $K \in \mathcal{K}$ and $T \in \mathcal{T}$, function $E_K(T, \cdot)$ is a bijection, with $E_K^{-1}(T, \cdot)$ denoting its inverse. The goal is that each tweak induces a permutation that looks random and independent from the permutations induced by other tweaks. Security may be formalized by the game defined in the left-hand panel of Figure 2.

A number of efficient, secure constructions of tweakable blockciphers are known. The simplest construction, due to Liskov, Rivest, and Wagner [LRW11] combines a hash function and a blockcipher. Let $h : \mathcal{T} \to \{0,1\}^n$ be a function sampled from a family of ϵ-almost-xor-universal hash functions. This means that for every $T, T' \in \mathcal{T}$, where $T \neq T'$, and $\Delta \in \{0,1\}^n$, the probability that $h(T) \oplus h(T') = \Delta$ is at most ϵ. (There are several well-known families of hash functions having this property.)

```
$\mathbf{Exp}^{\widetilde{\mathrm{sprp}}}_{E,b}(\mathcal{D})$
    $K \twoheadleftarrow \mathcal{K}$; $b' \twoheadleftarrow \mathcal{D}^{\mathbf{E},\mathbf{E}^{-1}}$
    return $b'$

$\mathbf{E}(T, X)$
    if $b = 1$ then return $E_K(T, X)$
    if $\pi_T = \bot$ then $\pi_T \twoheadleftarrow \mathrm{Perm}(n)$
    return $\pi_T(X)$

$\mathbf{E}^{-1}(T, Y)$
    if $b = 1$ then return $E_K^{-1}(T, Y)$
    if $\pi_T = \bot$ then $\pi_T \twoheadleftarrow \mathrm{Perm}(n)$
    return $\pi_T^{-1}(Y)$
```

```
$\mathbf{Exp}^{\mathrm{ind\$\text{-}otp}}_{E,b}(\mathcal{D})$
    $K \twoheadleftarrow \mathcal{K}$; $ct \leftarrow 0$; $\mathcal{E} \leftarrow \emptyset$; $b' \twoheadleftarrow \mathcal{D}^{\mathbf{OTP},\mathbf{OTP}^{-1}}$
    return $b'$

$\mathbf{OTP}(T, mode, m)$
    $ct \leftarrow ct + 1$; $X \leftarrow \textsc{frame}(mode, ct, m)$ // Outputs an OTP2 frame.
    if $b = 1$ then $Y \leftarrow E_K(T, X)$
    else $Y \twoheadleftarrow \{0,1\}^n$
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{(T, X, Y)\}$; return $Y$

$\mathbf{OTP}^{-1}(T, Y)$
    if $(\exists X)\ (T, X, Y) \in \mathcal{E}$ then return $X$
    if $b = 1$ then $X \leftarrow E_K^{-1}(T, Y)$
    else $X \twoheadleftarrow \{0,1\}^n$
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{(T, X, Y)\}$; return $X$
```

Fig. 2: **Left:** $\widetilde{\text{SPRP}}$ security and **Right:** IND\$-OTP security of tweakable blockciphers.

Let $E : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ be a standard blockcipher. The LRW tweakable blockcipher is defined by $\widetilde{E}_K(T, X) = E_K(X \oplus \Delta) \oplus \Delta$, where $\Delta = h(T)$. XORing $h(T)$ with the input ensures that, with high probability, the inputs of the blockcipher are different for different tweaks. The result is that blockciphers with different tweaks look independent to a computationally-bounded adversary.

The data frame. The protocol will use a tweakable blockcipher with arbitrary-length tweaks. (Many such constructions are known, cf. [LRW11,LST12].) Namely, let $\mathcal{K}$ be a finite set, $\mathcal{T} = \{0,1\}^*$, $n \in \mathbb{N}$ be a multiple of 8, $N = n/8$, and $E : \mathcal{K} \times \mathcal{T} \times \{0,1\}^n \to \{0,1\}^n$ be a tweakable blockcipher. The OTP2 frame is much simpler than the OTP frame, having only three components. It is designed to support multiple "modes of operation" used for protocols beyond one-time-password-based authentication. (We will discuss these shortly.)

- integer $X.mode$ ($X[0]$) — Specifies the mode of operation for the frame. This is used to signal to the authority which protocol is being executed.
- integer $X.ct$ ($X[1{:}4]$) — The 3-byte (24-bit) counter. This has the same semantics as $(token\_ctr, session\_ctr)$ in the OTP protocol; it is incremented each time the token is engaged.
- string $X.m$ ($X[4{:}]$) — The payload, comprised of $N - 4$ bytes.

The semantics of the payload and tweak are determined by the mode of operation; accordingly, whether the frame is deemed authentic depends on the mode. However, its freshness is determined by the $X.ct$ as before, and the authority must deem a non-fresh frame to be invalid. The format of the enciphered OTP2 is much the same as before, except that it depends on a tweak. Namely, it is computed as $otp = \langle I \,\|\, E_K(T, X)\rangle$, where T is the tweak and I is the identity of the token.

The token. The OTP2 hardware token will only do one thing: encipher data frames. The token takes as input the tweak T, an integer $mode$, and the payload m. When activated, it first checks that $|m| = N - 4$ and $mode$ is a valid mode. It then increments the internal counter, computes the data frame X, computes $Y \leftarrow E_K(T, X)$, and outputs Y. The token may be used in two ways. The first provides the no-client-side-code functionality of YubiKeys. Though it is inherently insecure when used for multiple services, we feel it should be supported. The second mode supports the full suite of OTP2-based protocols. The token has the following interfaces:

- $(otp, \sigma') \leftarrow \textsc{key2}(\text{"otp0"}, \sigma)$. Outputs a fully-formed OTP2 in $mode = 0$.
- $(Y, \sigma') \leftarrow \textsc{key2}(\text{"otp"}, T, mode, m, \sigma)$. Outputs an enciphered OTP2 data frame.
- $I \leftarrow \textsc{key2}(\text{"id"}, \sigma)$. Outputs the public identity of the token.

Each of these "calls" use the same state, so that $(otp, \sigma') \leftarrow \textsc{key2}(\text{"otp"}, \sigma)$ is equivalent to running

$$(Y, \sigma) \leftarrow \textsc{key2}(\text{"otp"}, \varepsilon, 0, [0]^{N-4}, \sigma);\ I \leftarrow \textsc{key2}(\text{"id"}, \sigma)$$

then computing $otp \leftarrow \langle I \,\|\, Y\rangle$. (See the OTP0 mode below.)

Physical considerations. An interface is *activated* either by the arrival of some inputs or by the client physically interacting with it. The “otp” interface is activated as follows. When a payload is waiting, the token starts blinking. Once the user *engages* the token by pressing the button, the payload is processed and an enciphered frame is output. Not every mode of operation requires engagement. The “otp0” interface is activated when there is no payload waiting and when the client engages the token. The “id” may be activated without engagement. (6.1: Chris says: Should the token somehow enable “cancelation?”)

6.1 Modes of operation

We describe four modes of operation for the token. (6.2: Chris says: This is the most interesting part from a cryptographic perspective. All of this looks “OK” to me, but each definitely needs a rigorous analysis.)

OTP0 ($\mathit{mode} = 0$). The simple, one-time-password mode. The payload consists of an all-zero byte string $[0]^{N-4}$ and the tweak is ε. Upon receipt, the authority deciphers the frame, checks that the payload is equal to $[0]^{N-4}$, and checks that the counter is fresh. This is the “no-client-side-code” mode that can be invoked via the token’s “otp0” interface. *It must be used with only one service*, and must be engaged by the client.

OTP ($\mathit{mode} = 1$). The *request-bounded, one-time-password* mode. Provides authentication of the client for a particular request. The request might be to login to a service, or for authorization to access a resource. The resource is encoded by the tweak. The payload is specified by the protocol, but it is a static value known to the client and authority. Upon receipt, the authority decrypts and checks that the payload matches the string it expects. Requires engagement.

Integrity ($\mathit{mode} = 2$). The *integrity* mode. This can be used to authenticate (i.e. MAC) a message. The payload is $[0]^{N-4}$ and the tweak is the message. Requires engagement.

The next two modes are used to implement *authenticated encryption with associated data*. Such a scheme provides privacy and authenticity of a message M and authenticity for associated data A. Normally the syntax requires an explicit nonce, but our scheme will not require a nonce and take advantage of the token’s stateful counter. It resembles the OCB mode of operation, but has some significant differences. The string $M \parallel [1] \parallel [0]^p$, where p is the smallest, positive integer such that $|M| + 1 + p \equiv 0 \pmod{N-4}$, is divided into $(N-4)$-byte blocks $(M_1, M_2, \ldots, M_\ell)$. These are XORed together to get the checksum R. Each block is processed in order in $\mathit{mode} = 3$ (defined below) to get $(C_1, C_2, \ldots, C_\ell)$. These blocks are XORed together to get S. Then $R \oplus S[:N-4]$ is processed in $\mathit{mode} = 4$ to get T. Finally, the ciphertext $C_1 \parallel C_2 \parallel \cdots C_\ell \parallel T$ is transmitted to the authority. The authority decrypts each ciphertext block as it arrives, checking that the counter value is larger for each block. (The counters need not be contiguous.) The last block is used to determine if the message is authentic. The authority must output the message only if it is authentic. The adversary deciphers the block (using A as the tweak), and checks that that the payload matches the checksum of the ciphertext and message blocks. (6.3: Chris says: Worth comparing this to https://eprint.iacr.org/2013/835.pdf.)

Transport ($\mathit{mode} = 3$). Processes a message block in transport mode. The payload is the message block M_i and the tweak is ε. This mode *does not* require engagement.

Transport Finish ($\mathit{mode} = 4$). Process the checksum of the message in transport mode. The payload is the truncated checksum $(R \oplus S)[4:]$ and the tweak is the associated data A. Requires engagement.

6.2 Security

The OTP2 is a building block for four modes of operation: simple authentication, request-bounded authentication, integrity, and transport. Each mode has a different security goal, and the hope is that the token is a powerful enough tool to achieve them. Of course, we need a proof of security for each mode. To that end, we have formulated a notion of security that models the token’s operation on the client’s side and the processing of the output on the authority’s side. The IND$-OTP game (right-hand side of Figure 2) is associated to a tweakable blockcipher, a bit b, and an adversary $\mathcal{D}$. The adversary is asked to distinguish the output of the token from a random string given given access to an oracle for the token’s “otp” interface. On input of a tweak, mode, and payload, if $b = 1$, then the oracle increments a counter, constructs and enciphers a frame using the provided tweak, and returns the output. If $b = 0$, it outputs a randomly chosen string. It is also given an oracle for deciphering frames. If $b = 1$, then it outputs the deciphered frame; otherwise it returns a random string.

Intuitively, security in the IND$-OTP sense implies that the OTP output of the token may be treated as a uniform random string. On the other hand, if the authority is asked to decipher an OTP not output by the token — or perhaps the OTP is valid, but the query involves the wrong tweak — then the deciphered frame will look random. We conjecture that these properties suffice to prove security of each of the modes of operation. (6.4: Chris says: This might be harder to prove than I think. AFAIK, OTP2 constitutes a **new security model**. The OTP2 is being used for different purposes simultaneously. I haven't seen any other crypto primitive used quite as flexibly as this.) However, we first need to prove the following:

Conjecture 1. *The* $\widetilde{\text{SPRP}}$ *security of tweakable blockcipher E implies the* IND\$-OTP *security of E.*

We suspect the proof follows from a similar argument used in Lemma 1. If so, we should get roughly birthday-bound security.

6.3 Applications

We present a few interesting applications in order to motivate the deployment of OTP2.

Two-factor authentication secure against dictionary attacks. *OTP mode* can be used to avoid sending a password hash in the clear via the following protocol. Let $\textsc{hash}$ be a cryptographic hash function with output length of $N-4$ bytes. The client $\mathcal{C}$ has a token $\textsc{key}2$ managed by authority $\mathcal{A}$ and a username-password pair (id, pw). The server $\mathcal{S}$ knows the identity I of $\textsc{key}2$ and $H = \textsc{hash}(id \,\|\, pw)$.

1. $\mathcal{C}$ choses a random, $(N-4)$-byte string R. It then computes $(Y, \sigma) \leftarrow \textsc{key}2(\text{“otp”}, 1, R \oplus H, J, \sigma)$, where J is the identity of the service $\mathcal{S}$. It then computes $I \leftarrow \textsc{key}2(\text{“id”}, \sigma)$, $otp \leftarrow \langle I \,\|\, Y\rangle$, and sends (R, otp, id) to $\mathcal{S}$.
2. $\mathcal{S}$ looks up the hash H associated with id, then computes $V = H \oplus R$ and sends (J, V, otp) to $\mathcal{A}$.
3. $\mathcal{A}$ decodes $\langle I \,\|\, Y\rangle \leftarrow otp$, looks up the key K and counter ct associated with I, then computes $X \leftarrow E_K^{-1}(J, Y)$. It checks if $X.ct > ct$ and $X.m = V$: if so, it lets $ct \leftarrow X.ct$ and sends $(J, otp, \text{“ok”})$ to $\mathcal{S}$; otherwise it sends $(J, otp, \text{“invalid”})$ to $\mathcal{S}$.
4. If $\mathcal{A}$'s response says “ok”, then accept; otherwise reject.

Each message sent between $\mathcal{S}$ and $\mathcal{A}$ is encrypted and authenticated using their shared key. (For example, they may use TLS.) This differs from the OTP (Section 4) and OTP1 (Sectoin 5) protocols, which only require these messages to be authenticated.

This protocol has the same round complexity as OTP and OTP1 and incurs only a bit of extra overhead. A key advantage of this protocol is that H is never sent in the clear, which prevents dictionary attacks on the password by $\mathcal{A}$. Dictionary attacks by a network adversary are prevented by using a secure channel (i.e. authenticated encryption) between $\mathcal{S}$ and $\mathcal{A}$.

Fine-grained resource authorization. Typically, web services have a *course-grained* model of authorization, meaning once you've provided credentials and logged in, you needn't provide your credentials again until you've logged out. There are notable exceptions, however. For example, GitHub requires you to provide your credentials before deleting a repository. To take another example, Amazon sometimes requires you to re-enter the last four digits of your credit card number in order to make a purchase. These exceptions motivate a need for *finer-grained* authorization for access to extra sensitive resources or to take certain actions. This is a perfect application for *OTP mode*.[8] The OTP1 protocol might be used for the same purpose. However, the tiny, 6-byte payload used their means collisions are quite likely.

Client authentication extension to TLS. In a typical TLS handshake, the server is authenticated to the client, but not the other way around. This leaves open the possibility of a man-in-the-middle. This looks like a job for *integrity mode!* Suppose the client and server are using ephemeral Diffie-Hellman. The ClientHello message contains the client's key share g^a. The client computes $(T, \sigma) \leftarrow \textsc{key}2(\text{“otp”}, 2, g^a, \sigma)$, then $I \leftarrow \textsc{key}2(\text{“id”}, \sigma)$, and sends $\langle I \,\|\, T\rangle$ along with the ClientHello. The server forwards g^a and $\langle I \,\|\, T\rangle$ to the authority, who computes $X \leftarrow E_K^{-1}(g^a, T)$ and checks that X is fresh and $X.m = [0]^{N-4}$. If so, it updates its state and tells the server to accept.

Key rotation. The *transport mode* provides a secure channel between the client and the authority. One way this can be used is to securely update the token's configuration. Currently, the only way to update the shared secret is to generate a fresh secret and physically copy it to the client and authority's systems. The transport mode could be used to *wrap* a key generated on the client's system, then transmit the wrapped key to the authority. The same technique can be used to update the counter, the public identity, and any other state that needs to be shared between the token and authority.

6.4 Concrete instantiations of the tweakable blockcipher

Our requirements for the underlying primitive are an infinite tweak space (i.e., $\mathcal{T} = \{0,1\}^*$) and a reasonable block size (e.g., 16 bytes). There are a number of constructions that would suit our needs, but we need to carefully consider the constraints

[8] Imagine it: you're about to delete a GitHub repo, and the website prompts you to stick in your token. Then a message pops up: “Are you sure you want to delete this?” Click “yes”: “Engage your token to confirm.”

of the token's hardware platform. YubiKeys already support AES-128 and HMAC-SHA1. It should be easy to show, under appropriate assumptions, that

$$\widetilde{E}_{K \,\|\, K'}(T, X) = E_K(X \oplus \Delta) \oplus \Delta, \text{where } \Delta = H_{K'}(T)[:16]$$

is $\widetilde{\text{SPRP}}$ secure, where E denotes AES-128 and H denotes HMAC-SHA1. The assumptions are that E is an SPRP and H is a PRF. This is essentially the LRW construction; the ϵ-almost-xor-universality is provided by the PRF security of H.

Since we're using algorithms already supported by the token, we suspect that OTP2 can be deployed without modifying the YubiKey hardware; only changes to the firmware should be required. One potential issue is that the secret key is now twice as long, since $|K \,\|\, K'| = 32$. (HMAC-SHA1 calls for a 16-byte key.) (6.5: Chris says: It depends on how the key is stored, I suppose. If it's impossible store larger keys, then perhaps we can design a new TBC that uses only a 16-byte key.) It's also worth noting that HMAC-SHA1 should not be considered a secure PRF, since SHA1 is known to be broken. (We have found a collision!) Therefore, it may be worth replacing SHA1 with SHA256.

Precomputing Δ. The transport and OTP0 modes use ε as the tweak. The value of $\Delta_\varepsilon = H_{K'}(\varepsilon)[:16]$ might be precomputed and stored on the token, thus avoiding an HMAC-SHA1 evaluation for these modes.

References

AS99. Anne Adams and Martina Angela Sasse. Users are not the enemy. *Communications of the ACM*, 42(12):40–46, 1999.

BHOS12. Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, IEEE S&P 2012, pages 553–567, 2012.

BR94. Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '93, pages 232–249, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

Eng08. Tobias Engel. Locating mobile phones using signaling service #7, 2008. Slides available at `https://events.ccc.de/congress/2008/Fahrplan/attachments/1262_25c3-locating-mobile-phones.pdf`. Accessed October 2017.

FID. FIDO. U2f v1.2 specifications. Available at `https://fidoalliance.org/download`. Accessed Octobor 2017.

Goo. Google. Google authenticator. Available at `https://github.com/google/google-authenticator`. Accessed October 2017.

Her05. Jonathan Herzog. A computational interpretation of Dolev-Yao adversaries. *Theoretical Computer Science*, 340(1):57–81, 2005.

Kam11. Dan Kaminksy. On the RSA SecurID comprimise, 2011. Available at `https://dankaminsky.com/2011/06/09/securid`. Accessed Oct 2017.

KS13. Robert Künnemann and Graham Steel. YubiSecure? Formal security analysis results for the Yubikey and YubiHSM. In *Security and Trust Management: 8th International Workshop, STM 2012, Pisa, Italy, September 13-14, 2012, Revised Selected Papers*, STM 2012, pages 257–272, 2013.

LR88. Michael Luby and Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, 1988.

LRW11. Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. *Journal of Cryptology*, 24(3):588–613, Jul 2011.

LST12. Will Landecker, Thomas Shrimpton, and R. Seth Terashima. Tweakable blockciphers with beyond birthday-bound security. In *Proceedings of the 32Nd Annual Cryptology Conference on Advances in Cryptology — CRYPTO 2012 - Volume 7417*, pages 14–30, New York, NY, USA, 2012. Springer-Verlag New York, Inc.

MBH$^+$05. D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen. HOTP: An HMAC-based one-time password algorithm. RFC 4226, 2005.

MMPR11. D. M'Raihi, S. Machani, M. Pei, and J. Rydell. TOTP: Time-based one-time password algorithm. RFC 6238, 2011.

MT79. Robert Morris and Ken Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, 1979.

ORP13. David Oswald, Bastian Richter, and Christof Paar. Side-channel attacks on the yubikey 2 one-time password generator. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 8145*, RAID 2013, pages 204–222, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

RSA. RSA. RSA SecureID hardware tokens. Available at `https://www.rsa.com/en-us/products/rsa-securid-suite/rsa-securid-access/securid-hardware-tokens`. Accessed October 2017.

RST$^+$16. Bradley Reaves, Nolen Scaife, Dave Tian, Logan Blue, Patrick Traynor, and Kevin R. B. Butler. Sending out an SMS: Characterizing the security of the SMS ecosystem with public gateways. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 339–356, 2016.

SSW16. Thomas Shrimpton, Martijn Stam, and Bogdan Warinschi. *A Modular Treatment of Cryptographic APIs: The Symmetric-Key Case*, pages 277–307. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

TZ17. Hacken Tanriverdi and Markus Zydra. Schwachstelle im Mobilfunknetz: Kriminelle Hacker räumen Konten leer. *Süddeutsche Zeitung*, May 2017.

Wag99. David Wagner. The boomerang attack. In Lars Knudsen, editor, *Fast Software Encryption: 6th International Workshop, FSE'99 Rome, Italy, March 24–26, 1999 Proceedings*, pages 156–170, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.