One-Time Authentication Code (OTAC)

A Technical Report
by
University of Surrey

Ioana Boureanu & Stephan Wesemeyer

July 15, 2021

Contents

Executive Summary

Findings

This report contains the functional analysis done by the University of Surrey (UoS) for swIDch, regarding one of their systems that generates a timed authentication code designed to identify and authenticate users registered with said system. The generated token is often referred to as an *"OTAC" (one time authentication code).*

The University of Surrey analysed swIDch's generic description of their OTAC system as well as two specific instantiations for two concrete use cases: the "epheremal" payment card and the drone use case.

The analysis pertained to whether the OTAC system and its use case specific instantiations]functionally satisfy the following high-level requirements:

- **Req 1: Identification.** The OTAC system provides identification and authentication of a given user.
- **Req 2: "Off-line" (Client) Operation.** The OTAC system operates off-line once commissioned, *i.e.*, no server/backend interaction prior to the identification & authentication process is required.
- **Req 3: Code-token Uniqueness.** A given generated OTAC token is unique per user, *i.e.*, given an OTAC token generated for user A, it must be impossible for the system to mistake it for an OTAC token generated by user B.

In our analysis, we demonstrate that the generic construction of the OTAC algorithm satisfies the stated requirements within the bounds of the specified system parameters. In fact,

- the generic OTAC algorithm provides identification of a given users: *i.e.*, the ability of a generic impersonator, as presented, to produce a valid OTAC token for a given user can be made as small as required.
- the system can operate well in a client "off-line" fashion, provided the time-synchronisation between the client application and the server application is maintained.

- The uniqueness of the OTAC values for each user is guaranteed by its construction.

Furthermore, we also demonstrate that the specific use case instantiations of the OTAC system for the "ephemeral" payment card use case and the drone use case, respectively, also meet these requirements within their specified system parameters.

Chapter 1

Introduction

1.1 Project Overview

This report has been prepared by the University of Surrey (UoS) in response to a request by swIDch to evaluate and analyse their One-Time Authentication Code (OTAC) algorithm/system. The main focus of this report is the analysis of the generic algorithm/system and whether it functionally satisfies a given set of requirements. Moreover, the report also looks at two specific use cases of the OTAC algorithm/system; these use cases are only looked at in terms of the adaptations they make to the generic OTAC algorithm and whether these changes do not hinder the specific requirements analysed for the generic OTAC algorithm.

1.2 Document Organisation

The report is structured as follows:

- Section 2 contains:
  - the **description of a generic time-sensitive authentication functionality** that the OTAC realises;
  - a **summative mathematical description of the OTAC specification** document that UoS received from swIDch; these specifications are quoted verbatim in Appendix B.
  - **three functional requirements, *Req 1*, *Req 2*, *Req 3* of the** OTAC.
  - Section 2.4 contains a **functional analysis of the OTAC against the three requirements**, stating the parametric/asymptotic correctness of these requirements, *i.e.*, with which probability, in terms of the parameters that define the OTAC, do these requirements hold if the OTAC is implemented and operates correctly.
- Section 3 describes two use cases of the OTAC system and functionally analyses whether the three main OTAC requirements are maintained by

these specific use cases, and to what concrete degree, *i.e.*, based on the specific parameters instantiated for these OTAC use cases, quantitative (not just asymptotic) measures of correctness are provided.

Appendix B contains the OTAC algorithm/system specification provided to UoS by swIDch.

Appendix A contains a supplementary refinement of Section 2.4 which looks beyond a purely functional analysis.

Chapter 2

OTAC – Overview & Analysis

The main focus of this section is the description of the generic OTAC algorithm/system, and a functional analysis of it against three given functional requirements.

2.1 Idealised Functionality of One-Time Authentication Tokens

We first describe the idealised functionality of a "One-Time Authentication Token" (see Figure 2.1). The OTAC algorithm is a specific realisation of this functionality, under specific requirements and guarantees. The three requirements of the OTAC system are given in Section 2.2

A system realising a "One-Time Authentication Token" would generally amount to a client application, C, which generates an authentication token/code[1] c, and a server application, S, which validates c. The client application C would need to be registered with the server S via a user ID (UID), u, and generally a user-specific secret, s. The client application would compute c using a function f which generally depends on this secret s, the UID u as well as the current time-interval ti (whose duration is configurable depending on the use case). For each such time-interval ti, a new computation of c is obtained. This time-sensitive token is intended to uniquely identify and authenticate the user associated with the UID u and secret s.

The token c is sent over the network to the server which applies the inverse of f to c, to retrieve the UID u. It then generally looks up the secret s corresponding to u in S's database and, using s, and a number of candidate time-interval values $ti_1, \cdots, ti_k$ for a configurable k, recomputes part of the token to see if it matches

[1]The terms token and code are used interchangeably in this report.

the received values. The need to use multiple candidate time-interval values is due to the possible time de-synchronisation between the client and the server.

If at least one of these checks pass, S will authenticate the user's credentials and potentially perform/initiate further actions for the user. These actions may differ from use case to use case and thus the token c sent over the network may have additional meta- and/or application-level meaning.

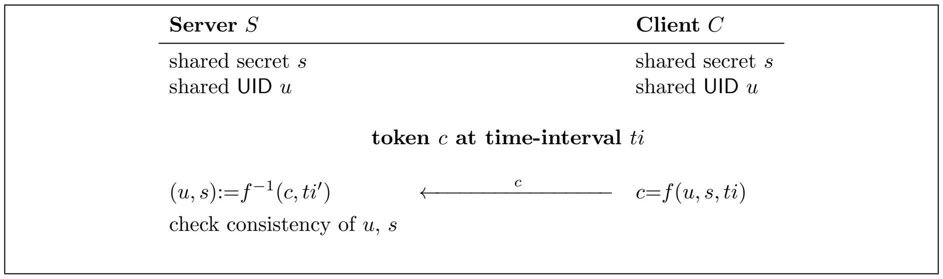


Figure 2.1: An Idealised Functionality F for One-time Authentication Tokens

Note 1: There are many ways to realise the idealised functionality F, under different assumptions, using different set-ups, and adhering to different requirements, *e.g.*, one can take a set-up where C and S do not use public-key, or one may require c to be of a specific length, or f to have given properties, *etc.*

In this vein, the OTAC system realises F and has its own requirements, set-up and assumptions. We will detail these in the next two sections below.

Most implementations of such functionalities do not actually authenticate a human presence (biometrically or by their active presence), instead they identify/authentication a user via/as their credentials s. The OTAC could be augmented to also biometrically authenticate a human user, but this is not in the scope of our discussions/analyses. As it is commonplace, we simply employ the word "user" to equate their registered secret s.

2.2 OTAC's Main Requirements

As we said, different functional requirements can be asked of a concrete system realising the idealising functionality F depending on that system's scope and the specifics of how F is realised.

The OTAC is a system that realises F (in a specific way described in Section 2.3) and has a specific set of functional requirements. In this report, we analyse the following three requirements:

- **Req 1: Identification.** The system is such that it should not be possible to impersonate a user, *i.e.*, without knowing the secret s, one[2] cannot generate a valid c for a given u.

[2] In our analysis, we are considering here a party following the system, even as an insider, with polynomial computational power but without active attacks onto the network, cryptography and, specifically, the registration/commissioning part of the system in which the user's secret s is transferred onto the client application.

- **Req 2: "Off-line" (Client) Operation.** Once the system has been commissioned (*i.e.*, users/clients are registered with the server), the system is such that the client application does not require any interaction or communication with the server/backend, in order to generate the token c and send it to the server.

- **Req 3: Code-token Uniqueness.** The function f is such that two honest users always generate different tokens c, *i.e.*, given a valid token generated with the OTAC system by user A, it must be impossible that the OTAC system mistakes this code for a valid code generated in the OTAC system by/for the user B.

This report analyses whether these three requirements are functionally met by the OTAC system as detailed in the specifications received by UoS from swIDch, and presented in Section 2.3.

Note 2: Other types of analyses, *e.g.*, with respect to different threat models for the network, the hardware and/or software are not considered. However, some details touching upon these aspects are analysed and included in Appendix A.

2.3 The OTAC Algorithm

In this section, we describe the specifics of how the OTAC system realises the idealised functionality F above. This mathematical description is based on the specification given to UoS by swIDch and found in Appendix B.

Just like the idealised functionality F, the OTAC system consists of a server-side application S and a client-side application C. The client-side application can be a mobile phone application or a stand-alone application. The communication channel between the client C and the server S is assumed to be a public channel with no extra provision of confidentiality, authentication or integrity. Note that while the generic case does not specify this, different use cases may require the transmission channel be encrypted, *e.g.*, the "ephemeral" Payment card use case discussed in Section 3.1.

We first introduce the OTAC system's objects and parameters.

2.3.1 OTAC System: Mathematical Objects & Parameters

a. u represent the unique user ID; the bit-length $|u|$ is a parameter of the system.

   This is often taken to be at least 6 decimal digits.

b. s represents the user-specific secret; the bit-length $|s|$ is a parameter of the system.

This secret is often assumed to be a pseudorandomly-generated bistring of 128 bits or longer.

c. ti represents a current time-interval; the duration/length of $|ti|$ is a system parameter.

   In practice, $|ti|$ can range from a few seconds to potentially days.

d. B_1, B_2 are system parameters defining different classes of tokens c; these B_1 and B_2 are mathematical bases and this defines the way in which the token c is represented, *i.e.*, which *characters* appear in c.

   Possible values for a basis B include

   - $B = 10$, in which case the *characters* appearing inside the token c are decimal digits $[0, \cdots, 9]$.
   - $B = 36$, in which case the characters appearing inside the token c are decimal digits and upper letters $[0, \cdots, 9, A, \cdots, Z]$.
   - $B = 62$, in which case the characters appearing inside the code c are decimal digits, upper & lower case letters – $[0, \cdots, 9, A, \cdots, Z, a, \cdots, z]$.

e. D_1, D_2 are system parameters that represent the number of characters to be used in the output.

f. other system parameters are derived from the above: *e.g.*, $n_1 \leq B_1^{D_1}$, and $n_2 = B_2^{D_2}$.

   In practice, B_1, D_1 are taken such that $n_1 >= 10^6$.

After a registration/commissioning process, both the client-side and the server-side applications have access to the user ID u and the user-specific secret s. Note that the allocation of the user ID, the secure generation of the secret s as well as the commissioning of these to the client and server applications are out of scope of this analysis. Similarly, the key management, *i.e.*, the secure storage and protection, of the secret s on the server and client applications is also not further analysed but assumed to be managed by appropriate hardware or software available on both client and server.

2.3.2 OTAC System: Code-computing Function f

Now, we describe how the code-computing function f given in the idealised functionality F is realised in the case of the OTAC.

Sub-procedures of OTAC's Code-computing Function f

- $TOTP(s, ti)$. This is a Time-base One-Time Password (TOTP) algorithm (*e.g.*, RFC 6238 [1]), based on a proven cryptographically secure hash function. Its output range is $[0, (n_1 - 1)]$, where $n_1 \leq B_1^D 1$.

  The outputs of $TOTP$ are assumed to be uniformly and independently distributed strings over the interval $[0, (n_1 - 1)]$; this is achieved provided $TOTP$ operates equivalently to the construction described in [1].

- $Merge_1(val_1, val2)$. The current description of this function computes $val_2 + val_1 \pmod{n_1}$, where $[0,(n_1-1)]$ is the range of the $TOTP$ function, and therefore a system parameter for $Merge_1$ too.

  – Please see the paragraph "OTAC's Code-computing Function f" for details on what arguments val_1 and val_2 are.

- $Merge_2(val'_1, val'_2)$. This function concatenates val'_2 with val'_1.

  – Please see the paragraph "OTAC's Code-computing Function f" for details on what arguments val'_1 and val'_2 are.

- $Spacing_i(x_i, D_i, B_i)$ for $i \in \{1,2\}$. This function transforms x_i into a string of length D_i using characters chosen from the set associated with the base B_i. So, D_i and B_i are system parameters.

  – Please see the paragraph "OTAC's Code-computing Function f" for details on what argument x is.

  – It is clear that in order to represent all possible values of x, it must be that $x \in [0,(B_i^{D_i}-1)]$.

  – For the server-side decoding part of the OTAC to work, the projection of the $Spacing_2$ function on the domain of x_2 must be invertible.

- $Distancing(val''_1, val''_2)$. This function simply adds the two values, *i.e.*, it computes $val''_1 + val''_2$ where $+$ is ordinary addition.

  – Please see the paragraph "OTAC's Code-computing Function f" for details on what arguments val''_1 and val''_2 are.

  – For the server-side decoding part of the OTAC to work, the projection of the $Distance$ function on the domain of val''_1 must be invertible.

We will now summarise the actual calculation of the OTAC token, *i.e.*, function f in Figure 2.1, as per the specification document found in Appendix B.

OTAC's Code-computing Function f

– **Step 1:**

(a) $hash_{val} := TOTP(s, ti) \in [0,(n_1-1)]$ this computes the TOTP using the shared secret s and the current time interval ti and returns a value in the interval $[0,(n_1-1)]$. Note that this is the *most* important function of the OTAC algorithm and the value of n_1 represents the number of distinct OTAC values that can be generated per user.

(b) $hash'_{val} = Merge1(hash_{val}, ti) = hash_{val} + ti \pmod{n_1}$.
Note that using modular arithmetic ensures $hash'_{val} \in [0,(n_1-1)]$: *i.e.*, it transforms a bitstring $hash_{val}$ output by $TOTP(s, ti)$ into an integer smaller than n_1.

(c) $C_1 = Spacing_1(hash'_{val}, D_1, B_1)$. This converts the $hash'_{val}$ into a prescribed format of length D_1 using the characters in base B_1.

For example, if $D_1 = 6$ and $B_1 = 10$ and $hash'_{val} = 123$, then one possible implementation of $Spacing_1$ could output $C_1 = 000123$, *i.e.*, 6 characters in base 10 encoding 123.

– **Step 2:**

(a) $dvalue = Distancing(u, C_1) = u + C_1$. This is normal addition in some basis.

So, prior to this standard addition, both u and C_1 will be converted to a common base.

(b) $C_2 = Spacing_2(dvalue, D_2, B_2)$. This converts $dvalue$ into a prescribed format of length D_2 using the characters in B_2.

– **Step 3:**

(a) $\mathsf{OTAC} = Merge_2(C_1, C_2) = C_2||C_1$. The final step in the algorithm is the concatenation of the two value computed in Step 1 and Step 2 to produce the overall OTAC value.

OTAC's **Server-side Decoding/Authentication** The value, OTAC, computed in Step 3, is then sent to the server where the inverse operations are carried out[3]:

– **Step 4: extract**

(a) extract C_1 and C_2 from $\mathsf{OTAC} = C_2||C_1$. This simply involves undoing the concatenation.

– **Step 5: retrieve u**

(a) compute $dvalue$ by applying the inverse of $Spacing_2$ to C_2, *i.e.*, $Spacing_2^{-1}(C_2, D_1, B_2) = dvalue$, which is possible as $Spacing_2$ is invertible.

(b) recover u by applying $Distance^{-1}(dvalue, C_1){=}dvalue - C_1{=}u$.

It is assumed that prior to the subtraction both $dvalue$ and C_1 will be converted to the common base used previously.

– **Step 6: verify using s**

(a) having retrieved u, the associated s is retrieved and k[4] candidate ti's are generated. These ti's are then used to compute candidate C_1's as

[3]We assume that the OTAC value has not been corrupted in transit, *i.e.*, the value sent by the client is the value received by the server.

[4]The value of k depends on the length of the chosen time invertal ti, the anticipated time de-synchronisation between the server and the client applications and the latency of the network over which the OTAC is sent. It is important to optimise the choice of k to ensure successful authentication of the client while minimising the number of time intervals to check.

described in Step 1 above. If $C_1 = C'_1$ for one of the ti's then the user u has been successfully identified and authenticated.

Observation with respect to $Merge_1$.

According to the specification provided, the rationale for this function is the attempt to reduce the likelihood of generating the same $hash_{val}$ at two different time intervals. However, this is still possible due to the fact that there exist values to satisfy the following equality:

$$TOTP(s, ti_1) + ti_1 = TOTP(s, ti_2) + ti_2 \pmod{n_1}$$

Note, however, from a security perspective, there are no issues with generating the same value for two different time intervals as long as these values are uniformly distributed over the range of the $TOTP$ function.

Note that this is the case for RFC6238 [1] and as the OTAC $TOTP$ function is assumed to be functionally equivalent to [1], $Merge_1$ could safely be removed without impacting the functionality of the OTAC algorithm.

2.4 Functional Analysis of OTAC's Requirements

We will now look at whether the OTAC algorithm described in Section 2.3 meets the requirements stated in Section 2.2, from a functional perspective (*i.e.*, when the system and the environment work correctly, or as expected).

2.4.1 Functional Requirement *Req1*: Identification

Recall that *Req1* states that it should be impossible for anyone[5] to impersonate a user with UID u, if they do not know the user's secret s.

The Impersonator's Capabilities. We assume that a potential impersonator, Imp, who would attempt to break *Req1*, knows u – as this can be easily obtained from an intercepted OTAC value. Furthermore, we assume that all remaining parameters of the OTAC construction are either known to Imp or can be easily reverse-engineered from the client application. Consequently, the only unknown parameter in an attempted illicit construction is the user-specific secret s.

From the description of the OTAC, it is clear that all functions except for $TOTP(s, ti)$ can be computed by Imp (as they do not depend on secret values such as s, and as they know the system).

The Impersonator's Probability of Success. Since the $TOTP$ function is assumed to be based on a cryptographically secure hash function, the values generated by it will not reveal any information about the secret s (*i.e.*, not to

[5]In our analysis, we are considering here a party following the system, even as an insider, with polynomial computational power and without active attacks onto the network, cryptography and specifically the registration/commissioning part of the system in which the user's secret s is transferred onto the client application.

a normal impersonator bounded by polynomial computations or information-theory bounds).

However, it is also clear that an impersonator does not need to know the secret s to generate well-formed OTAC values for a user u. Obviously, the impersonator can iterate blindly over the output domain of $TOTP$, and one of those values will be valid for a given ti.

If the $TOTP$ used generates outputs uniformly distributed, and crucially independent of one another (which the $TOTP$ in RFC 6238 [1] does), then the impersonator cannot even adaptively find an interval ti for which producing the correct $TOTP(s, ti)$ without knowing s gives him an advantage over another time-interval tj. Thus, for any time-interval, ti, the impersonator is standing roughly the same chance of producing illicitly a valid $TOTP(s, ti)$. This is stated as follows:

Statement 1: If the $TOTP$ used is as per the RFC 6238 [1], for any time-intervals ti, tj, $t_i \neq t_j$, given to an impersonator, Imp, or even chosen by them,

$$|Pr[Imp \text{ produces a valid TOTP} \,|\, for\, ti] - Pr[Imp \text{ produces a valid TOTP} \,|\, for\, tj]| \leq \epsilon_1,$$

where ϵ_1 is negligibly small[6].

Given statement 1, and the fact that there are a fixed number of n values for each $TOTP(\cdot, ti)$ spread uniformly, we can then prove the following statement:

Statement 2: If the $TOTP$ used generates outputs uniformly distributed over n distinct values, then $Pr[Imp \text{ produces a valid TOTP} | \text{for any } ti] = \frac{1}{n} + \epsilon_2$, where ϵ_2 is negligibly small.

On the server-side, the server will compute a number of acceptable OTAC codes (corresponding to each ti), in order to ensure that small a time drift will not prevent legitimate clients from authenticating. Let k be the number of server-acceptable/server-checked codes for a given ti.

Thus if there are k candidate OTAC tokens being generated on the server, then the likelihood of the impersonator submitting a correct one for a given time period is $\frac{k}{n}$.

Putting it all together, formally, we can prove the following statement:

Final Statement: *If the $TOTP$ used in the OTAC generates outputs uniformly distributed and independent of one another (as per e.g., the $TOTP$ in the RFC 6238 [1]), then the probability of an impersonator as described above to produce a valid TOTP for a user without knowing their secret, thus breaking Req1, is $\frac{k}{n} + \epsilon_3$, where n is the size of the domain of the $TOTP$ and k is*

[6] All the ϵ values in this subsection are negligbly small, and in practice they can safely be considered to be 0. They refer to two aspects: (a) the TOTP in REF 6238 or other TOTPs used are computationally secure function, *i.e.*, there is negligibly-small computational difference from one instance to another; (b) there are choices made in any OTAC's instantiation around and alongside the TOTP instance dependent on the system's parameters; one such set of choices versus another also entails negligibly-small computational difference.

the number of server-checked codes for one client-generated code, and ϵ_3 is a negligible amount.

Take-away Message: Provided that k/n above is sufficiently small (*e.g.*, $< \frac{1}{10^6}$) and that multiple incorrect submission attempts will result in the account being locked, then the requirement that an impersonator cannot forge a user's OTAC without knowing the user's secret s. Thus, in practice, under normal computational powers, and if the *TOTP* inside the OTAC is as per the RFC 6238 [1], then *Req1* is met.

2.4.2 Functional Requirement *Req2*: "Off-line" (Client) Operation

This requirement states that once the application has been commissioned and the user's details have been registered on both client and server side, the client OTAC application operates autonomous of the server side.

This requirement is clearly met as there is no communication between the server and the client application prior to the calculation of the OTAC value.

However, note that the application can only function in this "off-line" mode considering the three aspects below:

a. the duration/length $|ti|$ of the time interval ti;

b. there is latency l onto the network;

c. the synchronisation of the computer-clock/time on the client side and server side applications[7].

Let $|ti|$ be the duration/length of the time interval ti. Let τ be the communication time of any client-issued code (including in the network latency l), i.e., the time to get from the client to the server.
Let δ be the difference in time synchronisation between the client and the server's computer clocks.
Let k be the number of trials a server makes in checking a given OTAC, for a time interval.
Let opt_k be a function which computes the value k given δ, τ, $|ti|$ and n_1 (the size of the range of the TOTP), *i.e.*, $k{=}opt_k(n_1, \delta, \tau, |ti|)$.

Final Statement. *If the TOTP used in the* OTAC *uses client-server resynchronisation techniques when needed as per e.g., the TOTP in the RFC 6238 [1], then, for any given n, δ, τ, $|ti|$ as per the above, there exists an optimising function opt_k that selects $k{=}opt_k(n, \delta, \tau, |ti|)$ such that the server will accept a client's offline-produced* OTAC *with a probability $1 - \frac{k}{n}$ which is negligible close to 1, and so the requirement Req2 of the* OTAC *is met.*

[7]Addressing any potential de-synchronisation of the time between server and client application is out-of-scope of this analysis and it assumed that mechanisms are in place to prevent this.

Note 3: Such fine-tuning of server-side verification, in order to discriminate between honest/acceptable/legitimate behaviour (*i.e.*, correctness) and illicit/rogue/fraudulent behaviour (*i.e.*, security) is a known problem in computer science [2], which has practical solutions within given applications.Therefore, the OTAC's server-side application, to achieve good attainment of *Req2*, would need to be fine-tuned per use-case: with one acceptance threshold k for payments and another for drones, *etc.*

Take-away Message, In practice, this means that the server should be fine-tunable (with respect to the number of codes to check per given time interval, the TOTP range, the given application, *e.g.*, network latency) within reliable margins, such that, under normal network conditions for the usage, a client's offline-produced OTAC is almost always accepted, and hence, with such fine-tuning in place, the requirement *Req2* of the OTAC is always met.

2.4.3 Functional Requirement *Req3*: Code-token Uniqueness

Recall that this requirement informally states that two honest users always generate different codes c, *i.e.*, given an OTAC token generated by user A, the server will never mistake it for a valid OTAC token generated by/for user B.

The only part of the OTAC value containing the user's UID, u, is C_2. We can prove that the OTAC tokens will always be different. The proof is as follows.

Consider two 2 different users with UIDs u_1 and u_2, $u_1 \neq u_2$. Assume, by contradiction, that the users' OTAC values were the same, *i.e.*, the two $C_2||C_1$ values generated are the same for both users. More formally, we would have that $C_2^{u_1}||C_1^{u_1}{=}C_2^{u_2}||C_1^{u_1}$. From this we would have two aspects holding simultaneously:

$$C_1^{u_1} = C_1^{u_2} \tag{2.1}$$
$$C_2^{u_1} = C_2^{u_2}. \tag{2.2}$$

Let's take equation (2.2): $C_2^{u_1}{=}C_2^{u_2}$. From this, we would have, by the result of the *Distancing* function in Step 2 (*c.f.*, Section 2.3.1), that $u_1{+}C_1^{u_1}{=}u_2{+}C_1^{u_2}$. Since equation (2.1) must also hold, then[8] it follows that we must have: $u_1{=}u_2$. But, this is a contradiction with the working hypothesis. So, the the assumption in the proof is false: two codes cannot be the same for two different users. This concludes the proof.

Final Statement. *If the Distancing function projected on the user-domain is injective and the rest of the* OTAC *algorithm produces C_1 and C_2 as per the specifications in Section 2.3.1, then the* OTAC *produces unique codes per user, thus meeting functional requirement Req3.*

Take-away Message: Provided that the construction of C_2 in the OTAC (*i.e.*, the part of the code that contains the UID u) combines u with C_1 in a way that remains user-specific (*e.g.*, the Distancing algorithm does not truncate results

[8]Note that this assumes the addition of *Distancing* to be proper addition and not modular addition.

using modular arithmetic), then OTAC tokens generated per user are indeed unique to that user. In practice, this can be achieved easily.

Note 4. All this analysis disregards the threat of *replay attacks*; in this threat, an old client-generated code would be resent to the server and this code would be accepted by the latter. All applications not just the OTAC, which like the OTAC require "off-line" client operation (*i.e.*, no bidirectional active interactions between the client and the server) are prone to replays to some extent. More specifically for the OTAC system, if the "off-line" operation is maintained (as it is a main functional requirement here) and the current design of the OTAC-production remains unchanged (*e.g.*, one does not add further time-identification details such timestamps/counters or mechanisms such as public-key encryption), then one aspect that can improve OTAC's replay-protection is a server-side addition, as follows. The server could check that a code received for a given user and acceptable at the current point, has not been accepted before against the currently server-acceptable intervals.

2.5 Summary of Analysis

We have shown that the generic construction of the OTAC algorithm satisfies the stated requirements within the bounds of the specified system parameters. In other words:

- the generic OTAC algorithm provides identification of a given users: *i.e.*, the ability of a generic impersonator, as presented, to produce a valid OTAC token for a given user can be made as small as required, by choosing cryptographically secure TOTPs and fine-tuning the system parameters. A corollary of this is that an accidentally mistyped OTAC token might also be accepted with the same negligible probability.

- the system can operate well in a client "off-line" fashion, provided the time-synchronisation between the client application and the server application is maintained, and the server is well fine-tuned with respect to false rejection vs. abuse, in function of the system's parameters (*e.g.*, latency, time-interval).

- The uniqueness of the OTAC values for each user is guaranteed by its construction, if the C_2-creating function is user-specific (*i.e.*, mathematically, it is injective over the user-ids).

All of these can be relatively easily and reliably attained in practice.

Chapter 3

OTAC's Use Cases & Their Functional Analysis

In this section, we look at two specific instantiations of the OTAC algorithm described in Section 2.3; these are two concrete use cases. We first describe the motivation for the use case followed by providing an analysis of the main differences between the generic algorithm and its adaptation for the use case in question. In particular, we provide details of the constraints imposed on the choice of system parameters by the the use case. We then analyse whether each use case specific instantiation of the OTAC algorithm still meets the three main requirements stated in Section 2.2.

3.1 Use Case: "Ephemeral" Payment Cards

3.1.1 Use Case Description

The payment card use case can be summarised as follows: a user, identified via the identifier UID, registers her real payment card number with the swIDch server application which provisions the swIDch client application with a fresh "serial number" associated to said payment card. The swIDch server application also stores the user's card serial number; this card serial number is the OTAC-system shared secret s (see Fig. 2.1).

For the purpose of this use case, the swIDch algorithm as given in the specifications (see Appendix B) and presented herein in Section 2.3 is altered, such that the OTAC code-token c generated is a number that equates to a valid payment card number together with its expiry date and Card Verification Value (CVV) number. The purpose is the user enter (parts of) this code into an e-commerce website to pay for goods.

When presented to the payment gateway, the payment card number will need to be passed to the swIDch server application for decoding so that the user's id, UID, and associated real/secret payment card number (and its details) can be retrieved from the swIDch database to pay for the transaction.

The advantage of the solution is that e-commerce providers do not need to change their infrastructure[1] to support the swIDch solution as it produces valid payment card numbers. Moreover, due to the time-limited nature of the OTAC token, the temporary payment card number will expire in *e.g.*, 90 seconds and, hence, even if stolen in transit, it is of limited use to any attacker. Finally, the swIDch client application does not hold any real payment card details, either.

3.1.2 Adaptations of the Generic OTAC Algorithm

The following changes have been determined by analysing the provided specification in Appendix B. To describe these changes, we will mainly use our description in Section 2.3.

The main constraint[2] in this use case is the format of the final OTAC token which needs to match the following format:

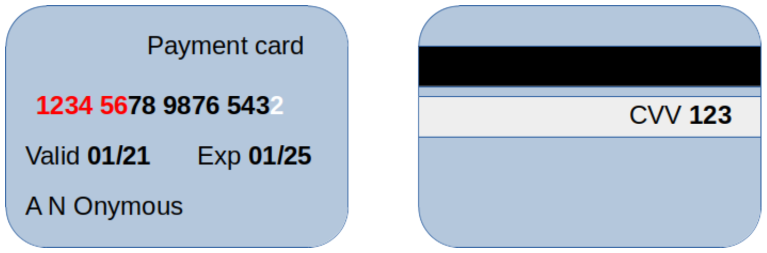


Figure 3.1: Payment Card Format

As a consequence, the main change in the credit-card OTAC is the construction of the final OTAC token which requires it to be a valid payment card number.

This imposes the following restrictions:

a. The payment card number consists of the Bank Id Number (BIN) which are the first 6 decimal digits and shown in red in Figure 3.1. The BIN is static and cannot be changed.

b. The next 9 digits denote the account ID ("789876543" in Figure 3.1) and can be freely chosen resulting in 10^9 possible values.

c. The final digit ("2" in Figure 3.1) is a check digit which is computed using the Luhn algorithm [3] and hence predetermined and cannot be chosen freely.

[1] This is so, provided swIDch can act as *e.g.*, a tokenisation service within the payment networks.

[2] Note that this equates to some of the parameters of the generic OTAC now being determined by a real-life use case; therefore, if nothing else changes, all probability thresholds in meeting the functional requirements given are dictated by usability/functional constraints.

d. The "Valid" and "Exp" dates can again be chosen freely provided the chosen dates are sensible, *i.e.*, they are within a reasonable time period and the "Exp" date is in the future of the current date while the "Valid" date is in the past. The swIDch algorithm restricts the 'Valid" and "Exp" dates to a five year period resulting in 60 possible values for these fields.

e. The CVV field is a 3 digit number which can be freely chosen providing 1000 possible values.

Functionally-imposed Restrictions: Number of Possible Codes. As a result of the above, there is a total of roughly $10^9 * 60 * 10^3 = 6 * 10^{13}$ possible "ephemeral" payment card numbers that can be constructed and that this use case OTAC can produce. So, formally, we have that $n = 6 * 10^{13}$.

New, Freely Made Design Choices. The main design decision for the payment card OTAC implementation is the number of customers that need to be supported. This is an important choice, since it determines how many codes/card-numbers per user, the system can provide. *i.e.*, if there were to be 10^7 customers then the algorithm would allow for up to $6 * 10^6$ possible "ephemeral" payment cards to be associated with each user.

So, it can be clearly seen that there is a clear security trade-off between the number of supported users and the number of "ephemeral" payment cards available per user.

Changes/choice of System Parameters

Let us re-fix the notation/details for the payment card use case. In this case, the construction of the OTAC algorithm uses the following settings:

- u represents the account ID, $u \in [0, 10^l]$ where 10^l is the number of customers to be supported;
- s represents a user-specific secret which is completely independent on the user's real payment card details;
- ti represents the current time-interval;
- $D_1 = 16 - l; D_2 = l$ are the number of characters to be used in the output format;
- $B_1 = B_2 = 10$ as all outputs are going to be digits;
- $TOTP(s, ti) \in [0, 6{*}10^{13-l})$ is as before but its output range is determined by the use case;
- $Merge_1$ does not change;
- $Merge_2$ is changed to $GenerateCardNumber(BIN, C_1, C_2)$ where BIN is a fixed 6 digit number identifying the bank and C_1 and C_2 are the outputs of Step 1 and Step 2 of the OTAC algorithm as described in Section 2.3;

- $Spacing_i(val, D_i, B_i)$ for $i \in [1,2]$ do not change
- $Distancing(val_1, val_2)$ has changed. It is uses $hash'_{val}$ computed in Step 1(b) of the generic algorithm instead of C_1.

The only two essential changes are the computation of *dvalue* and the replacement of $Merge_2$ by $GenerateCardNumber(BIN, C_1, C_2)$.

Firstly, the computation of *dvalue* uses the numeric value of $hash'_{val}$ before its transformation into C_1 to ensure that the resulting sum is not overflowing the range of the valid account numbers. Note that this also means that the maximum account number, u_{max}, that can be allocated is chosen such that $u_{max} + 6 * 10^{13-l} < 10^l$ in order to avoid a possible overflow.

Secondly, $GenerateCardNumber$ simply constructs a valid card number given the outputs C_1 from Step 1 and C_2 from Step 2 of the algorithm. The format of C_1 and C_2 are such that they represent the relevant 16 digts that can be freely chosen as discussed Section 3.1.2 with the constraint that the range of the "Valid" and "Exp" digits are valid and are sensible. Consequently, the only additional value that needs to be computed is the checksum digit.

We will now analyse these changes' impact on the requirements specified in Section 2.2.

3.1.3 Functional Analysis of OTAC*s* Payment use case

Functional Requirement *Req1*: Identification

This requirement is still as before: without knowing a serial-number register for a given user, an impersonator should not be able to produce a valid "ephemeral" card number as an OTAC token for them.

The question then is whether the only two changes to the generic algorithm give the impersonator an advantage. The identifying part of the OTAC is $C2$ and the hard-to-reproduce code (as discussed in Section 2.4.1) is $C1$, and specifically the TOTP inside $C1$.

In this transformation, the $Spacing_1$ function in the generic OTAC is replaced with the identity function, but original functionality of the $Spacing_1$ function is moved to this use case's $GenerateCardNumber$ function. So, overall no functionality is changed/lost between the generic OTAC and payment-centred OTAC. Aside of this, we can see that the payment-OTAC algorithm is identical in its construction of $C1$ and C_2 as the generic OTAC algorithm. The final step of the generic algorithm is the concatenation of C_1 and C_2, which is replaced by a concatenation of a BIN with C_2 and C_1 and the calculation of a check digit to form a valid payment card number. So the only difference here is some additional digits surrounding C_1 and C_2 but these do not change the purpose and content of C_1, C_2.

So, all the changes between the generic OTAC and payment-centred OTAC bring no functional differences, and are just about specifying specific parameters. Thus, we can recast the statement with respect to requirement *Req1* from the generic OTAC to this use case as follows.

Final Statement. In the payment OTAC, an impersonator will have a probability practically close to $\frac{k}{6*10^{13-l}}$ to impersonate a valid user without knowing their registration secret, where k is the number of server-checked codes for one client-generated code.

Consequently, we explained that the payment-based instantiation of the OTAC algorithm for payment cards is straightforward instantiation of the generic algorithm and hence satisfies this requirement *Req1* within its parameters.

Functional Requirement *Req2*: "Off-line" (Client) Operation

There is no change in the payment-centred instantiation of the OTAC algorithm that can affect this requirement in a sense not already discussed in Section 2.4.2. For this requirement to be attained, fine-tuning of k – the number of server-checked codes for one client-generated code, needs to occur in-keeping with this use case (*e.g.*, measurements of the quality of the connection of the client application can be envisaged, *etc.*). This is all practically attainable.

Functional Requirement *Req3*: Code-token Uniqueness

There is no change in the payment-centred instantiation of the OTAC algorithm that can affect this requirement in a sense not already discussed in Section 2.4.3.

Summary of the "Ephemeral" Payment use case Analysis

In this section, the use case for "ephemeral" payment cards has been presented and it was shown that the adaptation of the generic OTAC algorithm to this use case preserves the functional requirements stated in Section 2.2.

3.2 Use Case: Drones

This use case is the one derived from the algorithm description in Appendix B.

3.2.1 Use Case Description

Many drones are limited in their computational processing power and do not use encryption to secure the communication channel between the drone controller and the drone. As the drone commands are static, there is a real risk that an attacker can hijack a drone by simply sending appropriate drone commands to a drone within her range.

The proposed solution is to use the OTAC algorithm to encode the available drone commands together with a short time interval ti to ensure that the thus-obfuscated drone commands have a short validity period and, most importantly, an attacker will not be able to send valid commands in a given time interval (unless the attacker replays an already just-sent value), *i.e.*, property *Req1* of the OTAC is a crux requirement of the drone use case.

Functionally-imposed Restrictions: Number of Codes Sent per Second. Moreover, the solution provided here, needs to meet the following additional requirement:
– to be able to transmit a minimum of 20 commands per second.

Consequences of Functionally-imposed Restrictions. This aforementioned restriction in turn constraints of the length of the OTAC values that can be sent.

Moreover, even if the requirements *Req1*, *Req2*, and *Req3*, hold (as we will show below), this aforementioned restriction increases the threat around unencrypted channels and replay attacks (see Note 4 in Section 2.4). Yet, recall that this is not a problem specific only to this use case: in the generic case of the OTAC as well as in any systems with "offline" client communication and no encryption on the channel not just the OTAC, there is little replay protection (see, again, Note 4). However, in this use case, the extra constraint of sending a minimum of 20 commands per second means that an observer can potentially learn 1000 coded-commands within a 50 second window. Depending on the frequency with which these commands are sent and the usefulness of the commands being transmitted, an attacker might thus be able to obtain a useful set of commands in a very short period of time.

3.2.2 Adaptations of the Generic OTAC Algorithm

For this use case, the role of the OTAC algorithm changes from being used for the authentication and authorisation of a single user to a dynamic encoding system for a set of commands.

Using the notation introduced in Section 2.3, the idea behind this is as follows:

- A drone's $m \leq 1000$ commands are mapped to $1001, 1002, \cdots, (1000+m)$;
- Each of these numbers correspond to a UID, u_i;
- For each u_i, there exists a corresponding secret s_i;
- The client, in this case, is the drone controller and it is issued with a set of tuples $T = \{(u_1, s_1), (u_2, s_2), \cdots, (u_m, s_m)\}$ representing the mapping of the command code u_i to its corresponding secret s_i;
- The server, in this case the drone, is issued with the same set T.

With these assumptions in place, the generic OTAC algorithm is applied to issue a command, say u_x for some $x \in [0, \cdots, m]$, to a drone, using the corresponding s_x, the time interval ti and system specific parameters dependent on the available bandwidth of the channel between the controller and the drone.

Working Assumption. In the next analysis, we will only consider the implications of a single controller paired with a single drone. We thus assume that there is a one-to-one mapping between a controller and a drone. Moreover, we assume that while the set of commands $(u_1, \cdots, u_m)$ to control a drone can be identical in terms of their numerical values between drones, the associated set of secrets $(s_1, \cdots, s_m)$ are unique for each drone and that $|\{s_1, \cdots, s_m\}| = m$, *i.e.*, all the s_x are distinct.

3.2.3 Functional Analysis of OTACs Drone use case

In order to quantitatively analysis this use case, we will look at an actual instantiation of the OTAC algorithm for this use case (described in Appendix B) which deployed the following values:

- $D_1 = D_2 = 4$: the output size for the results of Step 1 and Step 2 of the OTAC algorithm is limited by the available bandwidth of the drone link.
- $B_1 = B_2 = 36$, *i.e.*, only capital letters and digits are allowed, however this means that up to $36^4 = 1,679,616$ different values are available.

Note that in order to avoid an overflow as a result of the *Distancing* function in Step 2 of the generic algorithm, the output of Step 1, needs to limited to $36^4 - (1000 + m)$, where m is the number of drone commands.

Functional Requirement *Req1*: Identification

In the context of this use case, this property is now interpreted to mean the correct identification of encoding for the command u_x, whichs was sent in a given time interval ti.

This holds trivially, following from the analysis of the generic algorithm and the lack of variation in this use case instantiation.

Functional Requirement *Req2*: "Off-line" (Client) Operation

We assume that the drone and controller are enrolled beforehand with the mapping of the commands u_x to their corresponding secret s_x.

Once thus commissioned, the "off-line" operation property for the drone use case holds as the instantiation of the OTAC algorithm is just a concrete implementation of the generic version which meets this requirement.

Functional Requirement *Req3*: Code-token Uniqueness

Since the instantiation of the OTAC algorithm in the drone use case is a concrete implementation of the generic version, it, too, meets this property.

Summary of the Drone use case Analysis

Considering, each command u_x individually, the OTAC algorithm for the drone use case satisfies all three requirements stated in Section 2.2.

Appendix A

Executive Security Analysis

In this appendix, we present a few notes on OTAC-related aspects, generally pertaining to security, which have not been covered in the main body of the document.

On Security Analysis vs Functional Correctness. Firstly, we note once again that the analysis in the main body of the document was carried out from the perspective of functional correctness and it is *not a full security analysis*. This is particularly true of requirements *Req2* and *Req3*. For requirement *Req1* we only considered a very specific and weak threat pertaining to impersonation. Namely, we did not differentiate between the cases of the attacker being an insider or an outsider, between whether she can or cannot corrupt client devices nor in which way she would choose the user to impersonate versus being asked to impersonate a given user. We did not quantify the attacker's success probabilities over the number of queries made to the system (of which type these queries would be, or at which frequency – given that the system has an acceptance window per code), *etc.*

On the Registration Process. We did not include the Registration Process in our analysis, but we stress that for the analysis to have positive results on the requirements holding, it is imperative that the users' secrets be generated pseudorandomly such that they uniformly distributed in their domain and independent of one another. In short, if this does not happen, the necessary conditions on the TOTP('s outputs) do not hold and the requirements of the OTAC would not be met.

On Specific Constructions/Design of the OTAC: Security & Usability. As we said before, the generic OTAC implementing the ideal functionality F in Section 2.1 could be replaced by other realisation of F, which still attain *Req1*, *Req2*, *Req3*. One such realisation is to employ public-key encryption and use the public key of the server to encrypt whatever data (identifiers, times) to be sent.

The advantage of this realisation of F is that provable security follows almost by construction, as does functional correctness of the requirements *Req1*, *Req2*, *Req3*. However, with the OTAC, what stops the designer from doing the above (*i.e.*, just sending an encrypted message) is the formatting and length constraints imposed by the use cases: *i.e.*, formatting the OTAC as a payment-card number or fitting within eight characters to encode drone commands. Nevertheless, for the payment-card use case, we note the following: In this day and age, all "one-time" or "long-term" payment-card numbers will be sent over an encrypted channel, like HTTPS. So, the worry of obfuscating the user-id in the OTAC becomes once again secondary. In fact, all that is needed is the formatting into an "ephemeral" credit-card number, which could be attain in the way the OTAC algorithm operates or others.

In this vein, a comparison with other services (*e.g.*, Revolut, Curve), as well as tokenisation services would be of interest from a security perspective.

Finally, depending on how *swIDch* is embedded or not into the payment networks (*i.e.*, *swIDch* sells this solution to a bank or *swIDch* remains a third party), the identification of the customer in/via the OTAC per se may not be always needed, as that information in the payment system (in each payment) is provided by various other sources (*e.g.*, the application cryptogram – AC, issued with each payment received by the issuing bank, *etc.*). In this case, the aforesaid point on just using the OTAC for formatting versus having a user-encoding OTAC stands further.

On Token Uniqueness. In the main body of this document, in the analysis of token uniqueness, we eliminated a corner case whereby the token of one user can be confused with the token of another user; this can occur functionally and not because of an attacker necessarily.

Indeed, note that there is always non-zero probability that an accidentally or deliberately corrupted OTAC value might result in a different user being authenticated as demonstrated here:

a. Assume that one OTAC$C_2 \,||\, C_1$ is corrupted to become OTAC $= C_2||C_1'$.

b. Then, let u' be the value $dvalue - C_1'$. This value u' may well be valid UID.

c. Then, by the argument presented in Section 2.4.1, there is a k/n probability[1] that the OTAC token is accepted for u'.

Note that for an attacker who knows the algorithm (which is always the case), and who, moreover, knows the user-space, it is even easier to perform step 1 and step 2 above: that is, to adaptively change C_1 into C_1' such that the resulting OTAC is a valid code for a user u'.

[1] $[0, (n-1)]$ is the range of $TOTP$ and k is the number of candidate ti' used by the server to allow for some time de-synchronisation between the client and server.

Appendix B

Specifications Received/Analysed by UoS

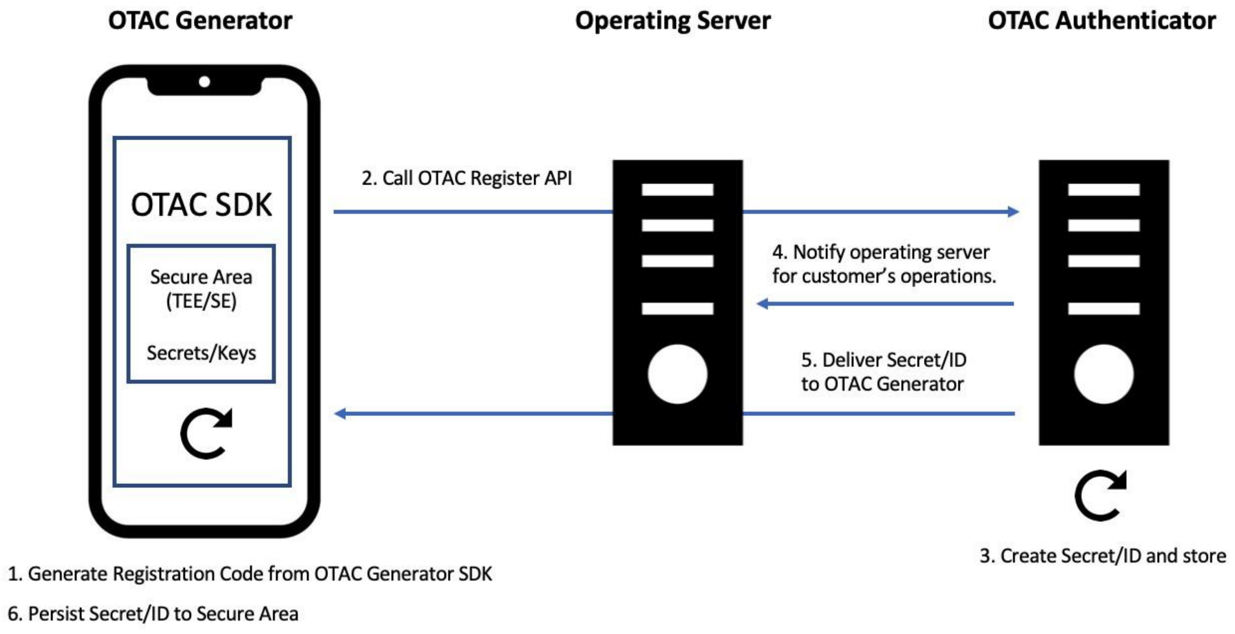
swIDch OTAC Algorithm Evaluation by University of Surrey

Registration

The manner by which the identifiers and secrets are provisioned to the client application is out-of-scope of the algorithm and it is assumed to be carried out as part of a trusted and secure commissioning process during the roll-out of the swIDch solution for a customer hence customer specific. There is nothing that can change the interaction of the algorithm.

We RECOMMEND that all the communications for registration SHOULD take place over a secure channel, e.g., Secure Socket Layer/Transport Layer Security (SSL/TLS) [RFC5246] or IPsec connections [RFC4301].

We provide a reasonable registration process that customers MAY apply to their system.


OTAC Generator
Operating Server
OTAC Authenticator
OTAC SDK
Secure Area (TEE/SE)
Secrets/Keys
2. Call OTAC Register API
4. Notify operating server for customer's operations.
5. Deliver Secret/ID to OTAC Generator
3. Create Secret/ID and store
1. Generate Registration Code from OTAC Generator SDK
6. Persist Secret/ID to Secure Area

- OTAC Generator is typically a user's mobile device where OTAC Client SDK is integrated to provide functionalities such as generating registration code or OTAC code.
- Operating Server is typically the customer's system which consists of a gateway to receive and relay the requests such as registration, and a system to process the customer's business logic.

- OTAC Authenticator is typically a server where OTAC Server SDK is integrated, often running within the same environment as the customer’s system, hence the customer is responsible for operating and maintaining OTAC Authenticator.

Secret

Secret (Key) SHALL be chosen at random or using a cryptographically strong pseudorandom generator properly seeded with a random value.

We follow the recommendations in [RFC4086] for all pseudorandom and random number generations. The following is an example of such system configurations.

- Length: 160 bits (recommendation), 128 bits (minimum length)
- RNG: NIST SP800-90A rev 1
- Entropy: The length of entropy SHOULD be the same strength as output key length (e.g. minimum 128 bits Entropy size).

We also ensure storing the secrets securely in the OTAC Generator and OTAC Authenticator system and, more specifically, encrypting them using tamper-resistant hardware encryption and exposing them only when required: for example, the key is decrypted when needed to verify an output value from f1 function, and re-encrypted immediately to limit exposure in the RAM to a short period of time.

Note that while we follow the security procedure explained above, the actual implementation to achieve the functionality of storing is dependent on the key management system within the platforms such as iOS or Android, and is out-of-scope of the algorithm.

ID (Unique Identifier)

ID is a unique identifier for a user or entity in a system, and de-identified value to make sure ID has no direct referencing to a given user or entity.

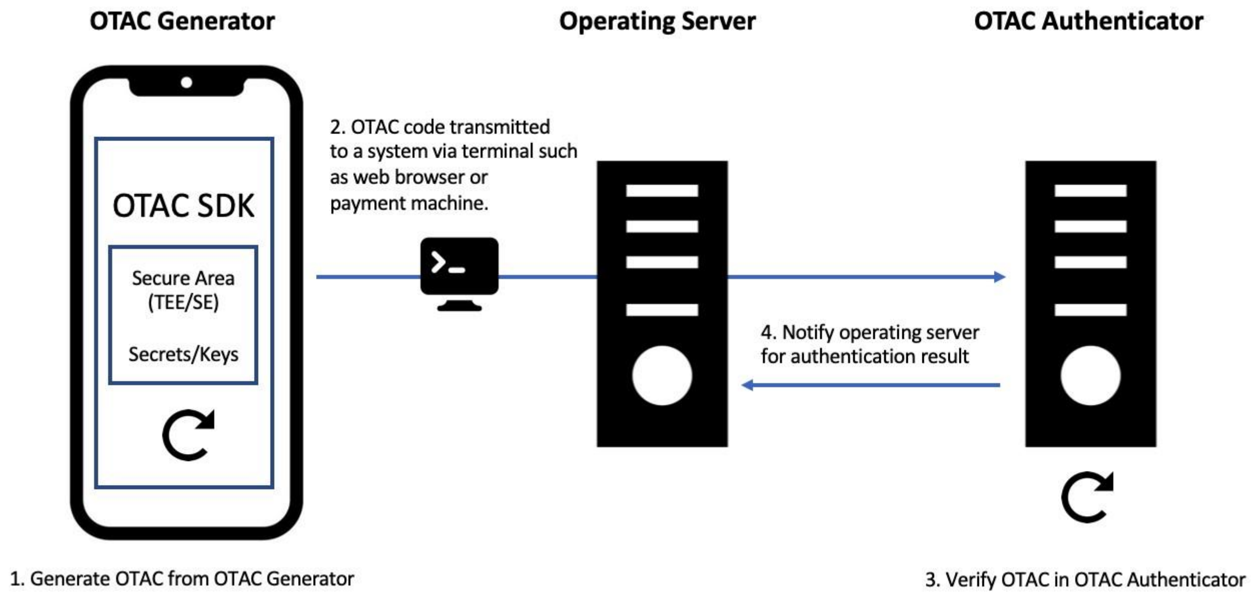
ID MAY be created and allocated as simple incremental values (e.g. 0000,0001,0002,0003…) or UUID [ISO/IEC 11578].

Since ID is a unique value, in conjunction with C1 value in f2, OTAC provides the notion of preventing duplicate between user or entity.

Generation and Authentication

Upon successful registration, OTAC Generator has the ability to generate OTAC without any prior communication to the server. The generated OTAC is then uni-directionally transmitted to terminals such as payment machines or another endpoint where usually the OTAC is routed to the customer's operating server. Note that the definition of terminals varies depending on use cases. The customer's operating server is responsible for delivering the OTAC to OTAC Authenticator where the OTAC is validated. The hosting environment of OTAC Generator and OTAC Authenticator SHOULD make sure the clock on both environments is synchronized, but this is out-of-scope of the algorithm, hence a requirement for the customer.

We provide a flow that customers MAY wish to apply to their system considering their business logic and use case.



OTAC Generator
Operating Server
OTAC Authenticator
OTAC SDK
Secure Area (TEE/SE)
Secrets/Keys
2. OTAC code transmitted to a system via terminal such as web browser or payment machine.
4. Notify operating server for authentication result
1. Generate OTAC from OTAC Generator
3. Verify OTAC in OTAC Authenticator

General OTAC Algorithm

Notations

I. C1 represents the output from f1.
II. C2 represents the output from f2.
III. K represents a shared secret between client (OTAC Generator) and server (OTAC Authenticator); each client has a different and unique secret K.
IV. T represents a time reference and a time step.
V. D1 and D2 represent the number of digits; system parameter, a big enough length to cover the output of the Spacing function

VI. B represents the Base value; system parameter

VII. U represents a unique identifier.

Generator

$$f1(K, T, D1, B) = Spacing1(Merge1(HASH(K,T) , T) , D1, B)$$

Description

- The primary purpose of f1 function is to produce C1 hence to add randomness to the computation of C1 and C2.
- The current choice of HASH function in f1 is TOTP [RFC6238] and MAY be replaced by another choice of HASH function that fulfils the same requirement of RFC6238.

1. hashval = Merge1(HASH(K,T), T)
   a. K and T are passed to the HASH function (HMAC Based as defined in RFC6238) in order to produce a TOTP [RFC6238] value and the output value is required to be of reasonable length considering user experience and the minimum value to be regarded as secure is 6 digits.
   b. Merge1 is a function to combine the output from HASH() and T to produce the hashval and the current choice of the operation is addition. As described in the associated patent, this is necessary in case TOTP value is duplicated, to ensure that the randomness stands still. The reasoning for having the Merge1 function is that in case there are duplicate TOTP values at two different time value T, for example, two TOTP values happen to be 839123 at two different moments and adding different T values to 839123 produces two different values to mitigate the situation, hence our patent contains this view.
   c. The length of hashval is a configurable parameter, and the length of hashval SHALL have enough space to cover the length of T and TOTP (whichever is longer) to support the Merge1 function. For example, if the length of TOTP value is 6 digits (the minimum value to be regarded as secure), and the length of T is 10 digits (e.g. Epoch timestamp), the space of hashval SHALL cover a minimum of 10 digits.
2. C1 = Spacing1(hashval, D1, B)
   a. The Spacing1 is a function to convert hashval to fit in a specific output format and size. The Spacing1 function is also important to maximize the numbers of codes with the same output size. The values of parameters D1 & B for Spacing1 function are governed by a set of customer's requirements and user experience.

An example scenario to choose D1 and B depends on use cases, but here's an example setting.

   i. Use case: Payment Card
   ii. OTAC Output Format: Decimal Digits
   iii. OTAC Output Length: 16 digits
   iv. Number of total users: 800k users
   v. The payment card number consists of the Bank Id Number (BIN, first 6 digits, static), the next 9 digits are the account ID, the last digit is the check digit, the next 4 digits are expiry date and the last 3 digits are CVC. OTAC consumes the 9 digits (account ID), 4 digits (expiry date) and 3 digits (CVC).
   vi. Therefore, one setting with one BIN number is B = 10, D2 = 6 and D1 = 10
   vii. Or another setting with 10 BIN numbers (e.g. each BIN covers 80k users) is B=10, D2 = 5 and D1 = 11.

b. D1 is a system parameter and it determines the final size of C1. Assuming the digits of C1 is D1, the Spacing1 function performs to make sure that the output value fits in D1.
c. B is a system parameter and a Base value that determines the representation of C1. The typical parameter values are 82, 62, 36 and 10. We RECOMMEND to have the highest value for B in order to maximize the probability in the same space and the Spacing1 function performs to translate into the chosen Base value. One thing to consider is that usability might depend on the choice of the Base value therefore the business requirement needs to be determined when designing the sensible combination of D1 & B. They output characters are as follows.
   i. B=10: digits only
   ii. B=36: digits+[A…Z]
   iii. B=62: digits+[A…Za…z]
   iv. B=82: digits+[A…Za…z]+20 other printable chars

f2(C1, U, D2, B) = Spacing2(Distancing(U, C1), D2, B)

Description

- The primary purpose of f2 function is to add the notion of distancing between U and C1 as described in the associated patent.

1. dvalue = Distancing(U, C1)
   a. U and C1 are passed to the Distancing function to produce a dvalue. U and C1 are always over the same domain when it comes to mathematical operation and it is to add U and C1.

2. C2 = Spacing2(dvalue, D2, B)
    a. The dvalue, D2 and B are passed to the Spacing2 which is a function to convert dvalue to fit in a specific output format and size. The Spacing2 function is also important to maximize the numbers of codes with the same output size.
    b. D2 is a system parameter and it determines the final size of C2. Assuming the digits of C2 is D2, the Spacing2 function performs to make sure that the output value fits in D2.
    c. B is a system parameter and a Base value that determines the representation of C2. The typical parameter values are 82, 62, 36 and 10. We RECOMMEND to have the highest value for B in order to maximize the probability in the same space and the Spacing2 function performs to translate into the chosen Base value. One thing to consider is that usability might depend on the choice of the Base value therefore the business requirement needs to be determined when designing the sensible combination of D2 & B.

OTAC = Merge2(C1, C2)

Description

- The final OTAC is then composed by putting C1 and C2 into the Merge2 function. The computational manner of the Merge2 function is C2||C1 currently.

Authenticator

The received OTAC is then split into C1 and C2, which gives an opportunity to reveal U by reversing the f2 function. Upon successful recovery of U, respectively the OTAC Authenticator performs the f1 function to derive a new C1. Upon completion of the computation, the received C1 and the newly computed C1 are compared and checked whether they match. If they match, OTAC Authenticator returns a successful result to the customer’s system to allow their business logic.

Payment OTAC Algorithm

Credit Card number format consists of the following format.

- BIN (6 digits, fixed number) || the account ID (9 digits) || the check digit (1 digit) || expiry date (4 digits) || CVC (3 digits)

Except for the fixed numbers, such as BIN and check digit, OTAC only utilises 16 digits in total comprising 9 digits for the account ID, 4 digits for the expiry date and 3 digits for CVC.

Notations

I. C1 represents the output from f1.
II. C2 represents the output from f2.
III. K represents a shared secret between client (OTAC Generator) and server (OTAC Authenticator); each client has a different and unique secret K.
IV. T represents a time reference and a time step.
V. D represents the number of digits; system parameter
VI. B represents the Base value; system parameter
VII. U represents a unique identifier.

Generator

$$f1(K, T, D1, B) = Spacing1(Merge1(HASH(K,T) , T) , D1, B)$$

Description

- Same as general OTAC except for D1 and B.

1. hashval = Merge1(HASH(K,T), T)
    a. Same as general OTAC
2. C1 = Spacing1(hashval, D1, B)
    a. Same Spacing1 function as general OTAC
    b. B is 10
    c. D1 is a system parameter and such system settings are suggested as follows in the configuration section below and the settings are customer requirements.

$$f2(C1, U, D2, B) = Spacing2(Distancing(U, C1), D2, B)$$

Description

- Same as general OTAC except for D2 and B.

1. dvalue = Distancing(U, C1)
    a. same as general OTAC.
2. C2 = Spacing2(dvalue, D2, B)
    a. Same Spacing2 function as general OTAC
    b. B is 10
    c. D2 is a system parameter and such system settings are suggested as follows in the configuration section below and the settings are customer requirements.

OTAC = GenerateCardNumber (BIN, C1, C2)

Description

- It depends how to fit C1 and C2 into the card format, and this is highly related to the configuration settings.
- One typical example would be the following.
    - BIN || C2 || checksum || C1
- Checksum is calculated only after the values are all concatenated and the Luhn algorithm is used to calculate Checksum. (https://en.wikipedia.org/wiki/Luhn_algorithm)

Configuration

We provide two sample configurations as follows and note that they are all system parameters.

9(the number of users) + 7(the probability) configuration

- The D of C2 is the first 9 digits: 10^9
- The D of C1 is the next 7 digits including expiry date (up to 5 years) and CVC : 5 x 12 x 1000 = 60000

7(the number of users) + 9(the probability) configuration

- The D of C2 is the first 7 digits: 10^7 1000000
- The D of C1 is the next 9 digits including expiry date (up to 5 years) and CVC : 5 x 12 x 100000 = 6000000

When such system is configured, we take a through discussion with the customer to make sure 'dvalue' never exceed 9 digits by considering the total number of users, hence the required digits for U and then 'Spacing2' for payment for example converts the ‘dvalue’ into whatever the format it needs to be. In the case that C1 is 9 digits, the 'hashval' is 5 digits and the Spacing1 will transform it to [2 digits of hashval][0-12][21-26][3 digits of hashval] to fit into the payment card number format.

BIN(6) || C2(7) || C1(first 2 digits) || checksum(1) || C1 (next 4 digits for expiry) || C1(remaining 3 digits for cvc).

In the payment term, BIN (Bank Identification Number, the first 6 digits of full card number) is a static information assigned to identify the institution that issues the card and the key in the process of matching transactions to the issuer of the charge card. The institutions may use one BIN or multiple BINs to make sure their service is capable of covering all of the users.

The OTAC configuration needs to be considered taking business aspects into account by asking how many users the customer intends to cover, and depending on the answer, the probability can be exploited.

For example, 1/6,000,000 is considered as a strong value for the probability to prove there is very little chance of duplicate, then the customer may choose to cover 10,000,000 users for one BIN number. If the customer is comfortable with projection to cover an even smaller number of users, then perhaps the customer may wish to go with another configuration to enhance the probability.

Therefore, the system parameters need to be configured considering the business aspect.

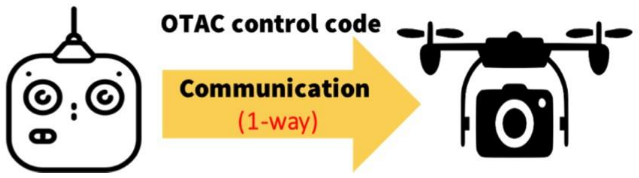
Drone OTAC Algorithm

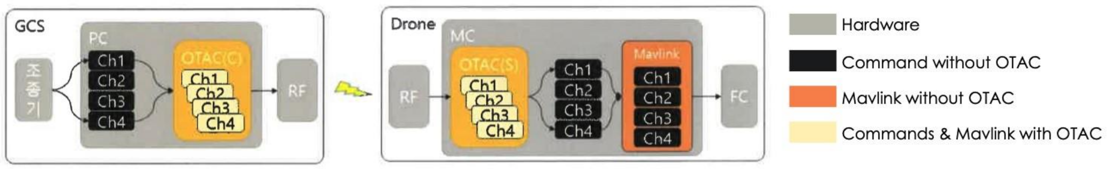
The Drone OTAC Algorithm is mostly derived from the General OTAC Algorithm described above. The noticeable variation is D and B for the Spacing.

We have done co-development with a military solution company in Korea and use this to elaborate the benefit of OTAC.

Technical background and current challenge

- Usually the control commands for drones are grouped from the controller(left) to the drone(right) over a protocol called Mavlink.





- Many drones are still constrained from having crypto hardware due to weight, performance, battery and cost issues, hence the need for a lightweight solution.

Radio Frequency 800-900 MHz (Long Range)

Min. 20 commands in a second

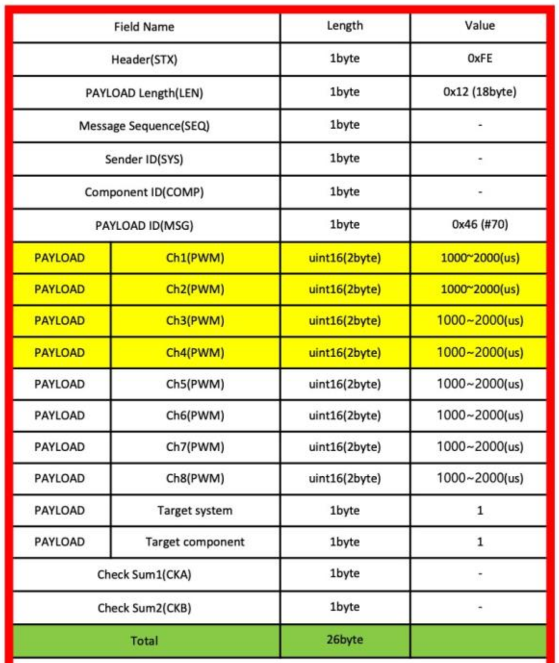
Dynamically secured communication

No extra hardware

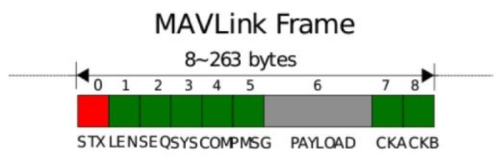
Lightweight

- Secured command has to be less than 200 bytes of data using Radio Frequency communication
- Send Min. 20 command codes in a second
- Dynamically secured communications
- Lightweight / No increase in weight of drone
- Not requiring extra hardware
- To work on top of CMVP protocol for military drones

- This creates a problem that the channel between the controller and drone is not secured, and the command values are static as shown below.



Field Name		Length	Value
Header(STX)		1byte	0xFE
PAYLOAD Length(LEN)		1byte	0x12 (18byte)
Message Sequence(SEQ)		1byte	-
Sender ID(SYS)		1byte	-
Component ID(COMP)		1byte	-
PAYLOAD ID(MSG)		1byte	0x46 (#70)
PAYLOAD	Ch1(PWM)	uint16(2byte)	1000~2000(us)
PAYLOAD	Ch2(PWM)	uint16(2byte)	1000~2000(us)
PAYLOAD	Ch3(PWM)	uint16(2byte)	1000~2000(us)
PAYLOAD	Ch4(PWM)	uint16(2byte)	1000~2000(us)
PAYLOAD	Ch5(PWM)	uint16(2byte)	1000~2000(us)
PAYLOAD	Ch6(PWM)	uint16(2byte)	1000~2000(us)
PAYLOAD	Ch7(PWM)	uint16(2byte)	1000~2000(us)
PAYLOAD	Ch8(PWM)	uint16(2byte)	1000~2000(us)
PAYLOAD	Target system	1byte	1
PAYLOAD	Target component	1byte	1
Check Sum1(CKA)		1byte	-
Check Sum2(CKB)		1byte	-
Total		26byte	



Notations

I. C1 represents the output from f1.
II. C2 represents the output from f2.
III. K represents a shared secret between client (OTAC Generator) and server (OTAC Authenticator); each client has a different and unique secret K.
IV. T represents a time reference and a time step.
V. D represents the number of digits; system parameter
VI. B represents the Base value; system parameter
VII. U represents a unique identifier.

Generator

f1(K, T, D1, B) = Spacing1(Merge1(HASH(K,T) , T) , D1, B)

Description

- Same as general OTAC.
- One example of D1 and B is described in the configuration section below.

f2(C1, U, D2, B) = Spacing2(Distancing(U, C1), D2, B)

Description

- Same as general OTAC.
- One example of D2 and B is described in the configuration section below.

OTAC = Merge2 (C1, C2)

Description

- The final OTAC is then composed by putting C1 and C2 into the Merge2 function. The computational manner of the Merge2 function is C2||C1 currently.

Configuration

The configuration for this project is as follows and note that they are all system parameters.

The project is designed to demonstrate that OTAC creates a dynamic/random code with a given ID (command value) on the controller without any connection, and the code is sent to the drone where it is identified and the command is revealed.

Usually the drones are assigned unique IDs, and may have the same or different set of command IDs depending on how to configure the system and map drones, but this is out-of-scope of the algorithm.

In this project, the IDs used in OTAC are associated with each command values in the range of 1000-2000.

4(the number of ID to cover 1000 different command values) + 4(the probability)

- The D2 is 4
- The D1 is 4
- The B of C1 and C2 is 36 (Base 36)
- U is a unique identifier for each command in the range of 1000 – 2000.
- Every U has a unique shared secret value K.

The consideration was that only 200 bytes can be carried over RF (Radio Frequency) and the controller already occupied 150 bytes. The estimate for OTAC size for each channel is 6~8 bytes, hence 24~32 bytes for 4 channels, making the total to be about 180 bytes.

Bibliography

[1] David M'Raihi, Salah Machani, Mingliang Pei, and Johan Rydell. Totp: Time-based one-time password algorithm. *Internet Request for Comments*, 2011.

[2] Christos Dimitrakakis and Aikaterini Mitrokotsa. Near-optimal blacklisting. *Computers and Security*, 64:110–121, 2017. doi: http://dx.doi.org/10.1016/j.cose.2015.06.010.

[3] Wikipedia. Luhn algorithm. URL `https://en.wikipedia.org/wiki/Luhn_algorithm`. Accessed: 2021-07-08.