METHOD AND SYSTEM FOR GENERATING TIME-BASED ONE TIME PASSWORDS

TECHNICAL FIELD

The present disclosure relates to time-based one-time passwords, and their use in various authentication applications, such as in consumer electronics and cyber-physical systems such as industrial control systems.

BACKGROUND

Availability or aliveness are critical to cyber-physical systems (CPS), especially to so-called critical infrastructure, such as power grids. It has been known for cyber attackers to shut down such infrastructure in denial-of-service attacks, or to disable certain services while hiding their malicious behaviours from being detected. For example, an attack may disable safety instrumentation systems (SIS) of an industrial control system, and if undetected, may cause significant safety risks. Thus, in an adversarial setting, knowing that a particular service such as a safety system is actually running is even more important than just running the service.

It would be beneficial to allow a device in the field to prove to a control centre that it is still "alive" and running. This is referred to as Proof of Aliveness, or PoA. One way to implement PoA is to send a heartbeat signal from the devices to the control centre every few seconds. However, an attacker in the network can easily intercept the heartbeat signal and keep replaying it to fool the control centre. It has recently been shown that a more sophisticated attacker can even simulate the normal behaviour of a compromised device and send the simulated data to deceive the control centre, while the actual device is not running at all.

Hence, a device should be able to send a proof message that can prove its authenticity; also, the proof should be different every time to prevent replay attacks. For example, remote attestation could be run repeatedly to realise a proof of aliveness scheme, but hardware-based attestation requires additional trusted hardware platforms, and software-based attestation requires the attestation program to be implemented in the most time-efficient way on a given hardware platform. The device has to disable its interrupts to satisfy the time optimality requirement of software-based attestation methods, but in the meantime, a real-time system will stop responding to the interrupt triggers during normal operations, which can possibly lead to catastrophic consequences, such as system failure. This implies that repeated execution of program attestation is not an ideal solution for

checking the aliveness of a CPS device. Furthermore, software-based program attestation is usually too computationally expensive to be called every few seconds.

Time-based one time password (TOTP) protocols share many similarities with a proof of aliveness protocol, such as single side authentication, single communication pass, and the efficiency requirement. TOTP protocols are widely used as a second authentication factor to enhance the security of mobile user authentication. During authentication, the prover/client sends a password generated based on a secret and the current time. After that, the password is sent to the verifier/server for verification using the present time and some pre-shared data.

Existing TOTP protocols are not suitable for application scenarios that are planned to run continuously for a long time, like a decade, in cyber-physical systems. This is because the proofs that can be generated according to such protocols are eventually exhausted. An instance of a PoA would then have to stop running and reinitialise itself, which is undesirable in cyber-physical systems.

SUMMARY OF THE PRESENT DISCLOSURE

In a first aspect, the present disclosure relates to a method for generation and/or authentication of time-based one-time passwords (TOTPs), comprising, at a client device:

performing an initialisation operation comprising:

determining a time period;

generating one or more one-way function (OWF) chains for each of one or more sub-periods of the time period, wherein an OWF chain for a sub-period is generated by:

dividing the sub-period into N segments of equal duration; and

recursively applying a one-way function N times to a random initial secret that is a head node of the OWF chain, to generate a tail node of the OWF chain; and

transmitting, to one or more verifier devices, an initialisation message comprising a client identifier and the tail nodes of respective OWF chains, such that a TOTP generated by the client device is verifiable by the one or more verifier devices using one of the tail nodes as a verify point.

Some embodiments further comprise performing a proof generation operation comprising:

determining a number M of segments remaining from a current time until the end of a current sub-period of the one or more sub-periods;

generating a TOTP by M recursive evaluations of the OWF, starting with a head node of the OWF chain for the current sub-period; and

transmitting an authentication message comprising the client identifier and the TOTP to one of the verifier devices.

In some embodiments, a head node of an OWF chain for each sub-period is generated by a pseudo-random number generator using a head node of an OWF chain for the immediately preceding sub-period.

In some embodiments, the method further comprises performing an auto-replenishment operation prior to all nodes of the OWF chain or OWF chains being consumed. For example, the auto-replenishment operation may comprise distributing one or more additional verify points to the one or more verifier devices using a one-time signature scheme.

Some embodiments comprise, at a verifier device of the verifier devices, performing an authentication operation comprising:

receiving the client identifier and the TOTP;

determining, based on a time of receipt of the authentication message, a number Z of time segments that have elapsed since a time of last authentication of the client device;

generating verification data by Z recursive evaluations of the OWF, starting with the TOTP; and

if the verification data matches the verify point for a current sub-period, updating the verify point to be the TOTP; and updating the time of last authentication to the time of receipt of the authentication message.

In some embodiments, the proof generation operation is performed at regular intervals as a proof of aliveness of the client device.

In some embodiments, the authentication operation further comprises determining whether a difference between the time of receipt of the authentication message and the time of last authentication of the client device exceeds an aliveness time threshold.

In a second aspect, the present disclosure relates to an apparatus for generation of time-based one-time passwords (TOTPs), comprising at least one processor in communication with computer-readable storage, the computer-readable storage having instructions stored thereon for causing the at least one processor to:

perform an initialisation operation comprising:

determining a time period;

generating one or more one-way function (OWF) chains for each of one or more sub-periods of the time period, wherein an OWF chain for a sub-period is generated by:

dividing the sub-period into N segments of equal duration; and

recursively applying a one-way function N times to a random initial secret that is a head node of the OWF chain, to generate a tail node of the OWF chain; and

transmitting, to one or more verifier devices, an initialisation message comprising a client identifier and the tail nodes of respective OWF chains, such that a TOTP generated by the client device is verifiable by the one or more verifier devices using one of the tail nodes as a verify point.

In some embodiments, the instructions further cause the at least one processor to perform a proof generation operation comprising:

determining a number M of segments remaining from a current time until the end of a current sub-period of the one or more sub-periods;

generating a TOTP by M recursive evaluations of the OWF, starting with a head node of the OWF chain for the current sub-period; and

transmitting an authentication message comprising the client identifier and the TOTP to one of the verifier devices.

In some embodiments, the instructions further cause a head node of an OWF chain to be generated for each sub-period by a pseudo-random number generator using a head node of an OWF chain for the immediately preceding sub-period.

In some embodiments, the instructions further cause the at least one processor to perform an auto-replenishment operation prior to all nodes of the OWF chain or OWF chains being consumed.

In some embodiments, the auto-replenishment operation comprises distributing one or more additional verify points to the one or more verifier devices using a one-time signature scheme.

The proof generation operation may be performed at regular intervals as a proof of aliveness of the apparatus.

A third aspect of the present disclosure relates to a verification server comprising at least one processor in communication with computer-readable storage, the computer-readable storage having stored thereon instructions for causing the at least one processor to perform an authentication operation comprising:

receiving a client identifier and a TOTP from a client device;

retrieving a stored verify point and a time of last authentication for the client device based on the client identifier and a current time;

retrieving a one-way function (OWF);

determining, based on a time of receipt of the authentication message, a number Z of time segments that have elapsed since the time of last authentication;

generating verification data by Z recursive evaluations of the OWF, starting with the TOTP; and

if the verification data matches the verify point for the current time, updating the verify point to be the TOTP; and updating the time of last authentication to the time of receipt of the authentication message.

In some embodiments, the authentication operation further comprises determining whether a difference between the time of receipt of the authentication message and the time of last authentication of the client device exceeds an aliveness time threshold.

In a fourth aspect, the present disclosure relates to non-transitory computer-readable storage having stored thereon instructions for causing at least one processor to perform a method as disclosed herein.

BRIEF DESCRIPTION OF THE DRAWINGS

Some embodiments of methods and systems for generating one-time passwords, for example in a proof of aliveness protocol, in accordance with present teachings will now be described, by way of non-limiting example only, with reference to the accompanying drawings in which:

Figure 1 is a block diagram of an example system for generating and verifying TOTPs;

Figure 2 shows flow diagrams of an example method for generating and verifying TOTPs;

Figure 3 is a schematic overview of the use of a one-way function (OWF) chain in the method of Figure 2;

Figure 4A is pseudocode of an example initiation procedure in a method for generating and verifying TOTPs that uses a OWF chain as in Figure 3;

Figure 4B is pseudocode of an example proof generation procedure in a method for generating and verifying TOTPs that uses a OWF chain as in Figure 3;

Figure 5 is pseudocode of an example proof verification procedure in a method for generating and verifying TOTPs that uses a OWF chain as in Figure 3;

Figure 6 is a schematic overview of the use of multiple one-way function (OWF) chains in the method of Figure 2;

Figure 7 is pseudocode of an example initiation procedure in a method for generating and verifying TOTPs that uses multiple OWF chains as in Figure 6;

Figure 8A is pseudocode of an example proof generation procedure in a method for generating and verifying TOTPs that uses multiple OWF chains as in Figure 6;

Figure 8B is pseudocode of an example proof verification procedure in a method for generating and verifying TOTPs that uses multiple OWF chains as in Figure 6;

Figure 9 is a block diagram of an example architecture of a client device; and

Figure 10 is a block diagram of an example architecture of a verification server.

DETAILED DESCRIPTION

Embodiments of the present disclosure relate to systems, devices and methods for generating and verifying time-based one time passwords (TOTPs), for example for use in a proof-of-aliveness protocol.

Typically, as shown in Figure 1, a system 100 for generating and verifying TOTPs comprises a server 110 communicating with a plurality of client devices 102.1, 102., ..., 102.N over a network 104. The server 110 acts as a verifier for TOTPs generated by the client devices. Although only a single server (verifier) 110 is shown in Figure 1, it will be appreciated that a number of servers 110 may be provided with functionality for verifying TOTPs sent over the network 104. This may be useful in the context of implementation of proof of aliveness protocols in cyber-physical systems such as industrial control systems, where some redundancy may be required in case a verification server becomes disabled, e.g. due to a cyberattack.

In some embodiments, the client devices may be consumer electronic devices such as smartphones, tablet computers, laptop computers and the like, while server 110 may be an authentication server that authenticates users of the client devices to enable the users to access services, such as email accounts and the like. In other embodiments, the client devices may be components of an internet of things (IoT) network, such as an industrial IoT (IIoT) network, and the server 110 may monitor the client devices by checking that it receives a proof of aliveness within a certain time interval, for example. The server 110 comprises, or is in communication with, a database 132 in which are stored data required for authentication, such as verify points and last authentication times for respective clients 102.1, ..., 102.N.

The following notation and definitions will be used in the present disclosure.

The security parameter is denoted by κ, an empty string by $\emptyset$, and the set of integers between 1 and n by $[n] = \{1,\ldots,n\} \subset \mathbb{N}$. If X is a set, then $x \xleftarrow{\$} X$ denotes the action of

sampling a uniformly random element from X. If X is a probabilistic algorithm, then $x \xleftarrow{\$} X$ denotes that X runs with fresh random coins and returns x. The binary representation of a value x with bit size n is denoted as $x = (x[1], x[2], \ldots, x[l_n]) \in \{0,1\}^{l_n}$.

A one-way function (OWF) is a function that is easy to evaluate but hard to invert. As will be understood by those skilled in the art, "easy" means that the function is able to be efficiently computed (typically within a few milliseconds or less), while "hard" means that given the output $y = OWF(x)$ of the function, it is impossible to obtain the image x in polynomial time. The input and output spaces of an OWF are denoted herein as $I_{OWF} = \{0,1\}^{l_r}$ and $R_{OWF} = \{0,1\}^{v \cdot l_r}$ respectively, where $l_r \in \mathbb{N}$ is determined by security parameter κ, and $0 < v \leq 1$. Embodiments therefore make use of length-preserving one-way functions that have an identical input and output space. A one-way function may be denoted by two algorithms OWF = (Setup, Eval):

- Setup(1^κ): The setup algorithm takes input 1^κ as the security parameter, and initialises an OWF and outputs the parameter pms which will be implicitly used by other algorithms. It will be understood that the content and usage of pms are dependent on the specific instantiation of the OWF. For example, the OWF may be realised with the hash function SHA-256, which has its own parameters pms, such as constant words and initial hash values that are used for computation of the hash using various logical functions. In some cases, pms may be NULL.
- Eval(x): The evaluation algorithm takes as input a value $x \in I_{OWF}$, and outputs a value $y \in R_{OWF}$.

The security of a strong OWF requires that, on given y = Eval(x) for an arbitrary x, it is hard to find its inverse. Embodiments of the present disclosure require the one-way function to provide the almost 1-1 property that each OWF value y only has one pre-image with overwhelming probability. An example of an OWF with the almost 1-1 property is the subset sum based OWF disclosed in R. Impagliazzo and M. Naor, *Journal of Cryptology* 9, 4 (1996), 199-216, the contents of which are incorporated herein by reference. This OWF can be proven to be secure in the standard model. In the random oracle model, the OWF may be instantiated with any collision-resistant hash function, such as a hash function in the SHA-2 family.

A Pseudo-random Generator (PRG) is a cryptographic primitive that amplifies the randomness from a short random seed into much longer 'pseudo-random' bit sequences. The input space of a PRG is denoted by $SS_{PRG} = \{0,1\}^{l_s}$ and the output space by $R_{PRG} = \{0,1\}^{l_g}$, where the bit-lengths $(l_s, l_g) \in \mathbb{N}$ are determined by κ. A pseudo-random generator may be denoted by two algorithms PRG = (Setup, Gen):

- Setup(1^κ): The setup algorithm takes as input 1^κ, and initialises the PRG and outputs the parameter pms which will be implicitly used by the other algorithm.
- Gen(s): The randomness generation algorithm takes as input a random seed $s \xleftarrow{\$} SS_{PRG}$, and outputs a pseudo-random value $r \in R_{PRG}$.

A proof of aliveness protocol is a process in which one party, client idC, proves its aliveness at a certain time to a server idS. A time period is divided into a set of discrete time slots $\{T_i\}$, each of which has a time length Δs, i.e., $T_{i+1} - T_i = \Delta s$. The time Δs is used to model the network delay while transmitting a proof, which will be considered as a global parameter. Let Δ_{rc} denote the life-span of a PoA protocol instance, and R_{PoA} denote the space from which a secret will be drawn, defined by κ. Let T_{att} be the aliveness tolerance time. Basically, if the server idS fails to receive a valid proof from the client idC within T_{att}, then idS would declare the death of idC.

A PoA protocol may be denoted by three algorithms Π = (Setup, NextPf, Verify) which are defined as follows:

- Setup(1^κ, ρ_{idC}, T_{start}, Δ_{rc}, T_{att}, aux): The setup algorithm takes as input 1^κ for the security parameter κ, the client idC's initial check-secret $\rho_{idC} \in R_{PoA}$, a protocol start time T_{start}, lifespan Δ_{rc}, aliveness tolerance time T_{att}, and an optional auxiliary parameter aux; and generates the system parameter pms, an initial aliveness verify-point π_{idC} and its checked time $T_{ack}^{idC} = T_{start}$. In some embodiments, the auxiliary parameter aux may indicate a number of sub-chains in a multi-chain proof of aliveness protocol (as will be described below). The first verify-point π_{idC} and its T_{ack}^{idC}, and pms are given to idS over a secure channel; whereas, idC stores the initial check-secret ρ_{idC} privately. Each client keeps a state variable $st_{idC} = (\rho_{idC}, pms)$; and the server idS has a state $st_{idS,idC} = (\pi_{idC}, T_{ack}^{idC}, T_{ack}, pms)$ for each client idC. In some embodiments, the states of the parties may be updated dynamically during a session. It is assumed that the parameters in pms will be implicitly used by other algorithms.
- NextPf(st_{idC},T): The proof generation algorithm takes as input the current state st_{idC} of the client idC and a time slot T as input, and outputs the proof x that should be sent out at time T. The state st_{idC} may be updated along with the invocation of this algorithm.
- Verify($st_{idS,idC}, x, T, T_{ack}^{idC}$): The proof verification algorithm takes as input the current state $st_{idS,idC}$ (at time T) used to verify the aliveness of the client idC, a proof x, and a time slot T as input, and outputs 1 if x is valid for T and $T - T_{ack}^{idC} \leq T_{att}$, and 0 otherwise. In the meantime, the verify-point π_{idC} and T_{ack}^{idC} may be updated to the new ones, i.e., $\pi_{idC} \coloneqq x$ and $T_{ack}^{idC} \coloneqq T$, if this algorithm returns 1.

Embodiments of a method of verifying proof of aliveness will now be described with reference to Figures 2 to 8.

Figure 2 shows an overall process flow for an initialisation process 200A and a proof generation and verification process 200B performed by a client device idC (such as client device 102.1 in Figure 1) and a server idS (such as server 110 in Figure 1). Each of the embodiments discussed below is based on the use of OWF chains spanning a given time period to generate and verify proofs. In some embodiments, a single OWF chain may be used for the entire time period, while in other embodiments, a plurality of OWF chains may be used. An advantage of using multiple OWF chains according to certain embodiments is that it enables auto-replenishment of proofs as will be described in more detail below.

The initialisation process 200A comprises the client idC determining, at step 202, a time period for the proof of aliveness protocol, defined by start and end times T_{start} and T_{end}.

Next, at step 204, the client idC generates an OWF chain for the time period, or for each sub-period of the time period, if multiple OWF chains are to be generated. In the latter case, the OWF chains for successive time periods are linked by setting the start time of a chain to the end time of the previous chain in the sequence. The OWF chain for a time period is generated by N recursive applications of an OWF (which may be an almost 1-1 OWF as discussed above), where N is the number of time segments in the time period or sub-period covered by the OWF chain. The result of the last application of the OWF is a tail node of the OWF chain.

At step 206, the client idC stores the head node of the OWF chain, or of the first OWF chain if there is more than one. The head node is a secret value that is subsequently used by the client idC to generate proofs. In embodiments that use multiple OWF chains, as will be explained further below, a seed for a pseudo random number generator may also be generated for each OWF chain during step 204, and if so the seed for the first chain is also stored at step 206.

At step 208, the state of each OWF chain is transmitted to the server idS, for use in verifying proofs that are subsequently generated by the client idC. The state of an OWF chain comprises the tail node of the OWF chain (i.e., the initial verify point), and may comprise other information such as the checked time of the initial verify point ($T_{ack}^{idC} = T_{start}$), the end time of the time period or of each sub-period, a last verification time, and a number of sub-periods.

At step 210, the server 110 stores, in database 132, the states of the respective OWF chains for use in subsequent proof verifications.

A high-level flow of a proof generation and verification process 200B is also shown in Figure 2. Process 200B comprises the client idC determining, at step 220, which OWF chain to use based on the current time. If there is only one OWF chain, this step is not performed.

Next, at step 222, the client idC generates a proof based on the current time. In some embodiments with multiple OWF chains, step 222 may also comprise determining a seed value for a pseudo random number generator so that the head node for generating the OWF chain may be determined.

At step 224, the proof is sent to the server idS, which verifies it at step 226 using the state information for the OWF chain for the current sub-period.

In a proof of aliveness protocol, the process 200B may also comprise waiting a predetermined amount of time, at step 228, and then returning processing to step 220 to generate the next proof to send to the server idS. For example, the predetermined amount of time may be $c \cdot \Delta s$ where Δs is the length of one time segment and c is a positive integer.

Turning now to Figures 3 to 5, a first example of a PoA protocol will be described. The PoA protocol of Figures 3 to 5 is based on length-preserving strong one-way functions (OWF) with almost 1-1 property, and is referred to herein as Π_{OWF}. A schematic overview of the construction of Π_{OWF} is shown in Figure 3. The PoA protocol has a chain structure in which each internal chain node (except for the tail) is a proof which can be used to prove the aliveness of the client to the server at the corresponding time. If the verifier receives at least one valid proof within the aliveness tolerance time T_{att} starting from T_{ack}^{idC}, then it is convinced that the client is alive. The nodes of the UWF chain are consumed in a direction from the tail to the head of the chain.

In the specific example shown in Figure 3, the most recently valid proof received by the server is used as the verify-point $\pi_{idC} = x_{N-1}$ to verify the authenticity of the next proof. The server accepts the proof x_T (for $T = T_{end} - 2\Delta s$) if it results in π_{idC} by (N-3) recursive one-way function evaluations.

Pseudocode for example algorithms of the protocol Π_{OWF} are shown in Figures 4A, 4B, and 5.

In an initialisation process (corresponding to process 200A of Figure 2), each client idC chooses a random initial secret $\rho_{idC} \xleftarrow{\$} I_{OWF}$ and runs Π_{OWF}.Setup(1^{κ}, ρ_{idC}, T_{start}, Δ_{rc}, T_{att}, $\emptyset$) (Figure 4A) to initialise an OWF-chain which uses a one-way function F that is uniformly selected from a family of almost 1-1 OWF functions, by invoking $pms_{idC} \xleftarrow{\$} F.Setup(1^{\kappa})$. The OWF-chain starts from a head node $x_0 = \rho_{idC}$ and ends at a tail node x_N, where $N = \left\lfloor \frac{\Delta_{rc}}{\Delta s} \right\rfloor$ is the length of the chain, and each node x_i for $1 \leq i \leq N$ is computed as $x_i := F.Eval(x_{i-1})$.

Eventually, the head node $\rho_{idC} = x_0$ is privately stored by the client as the authentication secret. The tail node x_N is sent to the server idS, which is set as the initial verify-point $\pi_{idC} = x_N$ used to verify the proof (i.e., any node x_i for $1 \leq i \leq N$). The first verify-point may be handed over to all potential verifiers via an authenticated channel. For a PoA protocol, the value of T_{att} is chosen such that $T_{att} > c\Delta s$ for some constant $c \in N$, where c is determined by the demands of specific systems.

In a proof generation process (corresponding to steps 220, 222 and 224 of the process 200B of Figure 2), the client idC runs the proof generation algorithm $\Pi_{OWF}.NextPf(st_{idC}, T)$ (Figure 4B) to obtain the proof x_T for the time slot $\coloneqq \frac{\lceil T_{end} - current\ time \rceil}{\Delta s}$. Note that all OWF-chain nodes can be computed based on x_0 thanks to the chain structure by design. Then, client idC can prove its aliveness by sending the credential x_T to the server idS.

In a proof verification process (corresponding to step 226 of the process 200B of Figure 2), upon receiving a proof $x = x_T$ of the client idC, the server idS verifies it by checking whether the difference between the receiving time T and the last aliveness check time T_{ack}^{idC} is smaller than the aliveness tolerance time T_{att}, and also whether the current verify-point $\pi_{idC} \in st_{idS,idC}$ can be computed from x_T (for example, as defined by the algorithm $\Pi_{OWF}.Verify$ of Figure 5). If $\Pi_{OWF}.Verify(st_{idS,idC}, x_T, T, T_{ack}^{idC}) = 1$, server idS accepts the client's proof and updates the verify-point by setting $\pi_{idC} \coloneqq x_T$ and $T_{ack}^{idC} \coloneqq T_{ver} \coloneqq T$. Note that, in the definition of $\Pi_{OWF}.Verify$ in the embodiment of Figure 5, the time T is implicitly checked by specifying the number Z regarding the distance from the time T to the last verified time T_{ver}. If the client and the server have an unsynchronized time T, then the verification will fail as well.

In some embodiments, each party may uniformly select the OWF function (from the OWF family) to prevent pre-processing attacks for inverting the function. Furthermore, to instantiate the OWF with a public function (such as SHA-256), each client could (implicitly) feed the function with a uniform salt, i.e., OWF.Eval(·) = SHA-256(salt , ·).

Some embodiments provide a PoA protocol based on pseudo-random generators (PRG) and one-way functions (OWF). This protocol is referred to herein as Π_{OWF}^{PRG}. In these embodiments, the PoA construction enables auto-replenishment. This avoids the need to use out-of-band mechanisms to securely send the tail OWF-chain node (i.e., the initial verify-point) of Π_{OWF} to replenish proofs, thus avoiding a high maintenance cost for both the client and the server. Auto-replenishment is important for deploying PoA in a CPS that requires unlimited proofs without interruptions. Further, the advantage of an adversary is highly related to the length of the chain; i.e., the longer the chain is, the larger parameters are required. Auto-replenishment enables use of a shorter chain that can use smaller parameters, thus improving security while also improving computational performance. A

shorter chain also requires fewer evaluations of the OWF for proof generation and verification, this also leading to improved performance.

To provide auto-replenishment and avoid performance bottlenecks, multiple OWF-chains in Π_{OWF} instances may be used. The life-span Δ_{rc} of a PoA protocol instance may be split into η time periods, where $\eta \in \mathbb{N}$.

In some embodiments, in each period, an independent OWF-chain of Π_{OWF} may be used. At the initialisation phase, the client stores all OWF head nodes and the server stores (in database 132, for example) all OWF tail nodes. Then the client and the server can choose the corresponding head and tail for authentication and verification respectively based on the current time. This leads to better computational performance if each OWF-chain is not too long. However, this solution will bring about $O(\eta)$ extra storage overhead to both the client and the server.

Since the client might be storage-constrained, in other embodiments, the storage cost for the client can be further improved from $O(\eta)$ to $O(1)$. This is done by leveraging a PRG to generate the head nodes for the OWF-chains, as shown schematically in Figure 6. Note that a PRG is typically less efficient than an OWF, so in at least some embodiments, the structure of the protocol is kept as simple as possible, i.e., only two layers may be used (rather than a multi-layer tree structure).

The protocol of Figures 6 to 8 makes use of a pseudo-random generator G = (Setup, Gen) and the PoA protocol Π_{OWF} = (Setup, NextPf, Verify) of Figures 3 to 5. Here, G is initialised with a range such that $R_{PRG} = SS_{PRG} \times I_{OWF}$. In some examples, the pseudo-random generator may be based on AES. It will be appreciated that many other implementations are also possible.

The pseudocode of example algorithms for Π_{OWF}^{PRG} is shown in Figures 7, 8A and 8B.

In an initialisation process (corresponding to process 200A of Figure 2), each client idC divides the life-span (time period) Δ_{rc} of the protocol into η sub-periods each of which has time length $\Delta'_{rc} = \lfloor \Delta_{rc}/\eta \rfloor$, where η is a parameter determined by the instantiation of the PRG (as discussed below). For the i-th ($i \in [\eta]$) time period, client idC runs the pseudo-random generator G to compute the corresponding initial secret ρ_{idC}^{i} and the PRG seed ss_{idC}^{i} (for computing the OWF-chain head of the next time period), i.e., $ss_{idC}^{i} || \rho_{idC}^{i} := G.Gen(ss_{idC}^{i-1})$. That is, the first l_s bits of the output of $G.Gen()$ form the seed for the next time period, while the remaining l_r bits form the initial random secret for the next time period. Client idC also initialises the i-th OWF-chain by running the setup algorithm of Π_{OWF} (Figure 4A), i.e., $\Pi_{OWF}.Setup(1^{\kappa}, \rho_{idC}^{i}, T_{end}^{i-1}, \Delta'_{rc}, T_{att}, \emptyset)$, where the start time T_{start}^{i} of the i-th OWF-chain is the end time T_{end}^{i-1} of the $(i-1)$-th OWF-chain, and $T_{end}^{1} = T_{start}$. The

returned client state is $st_{idC} = (ss_{idC}^{u}, \rho_{idC}^{u}, \{T_{end}^{i}\}_{0 \le i \le \eta}, \eta, u, pms)$. This only includes the first OWF-chain head ρ_{idC}^{1} and PRG seed ss_{idC}^{1}. The returned server state is $st_{idS,idC} = (\{st_{idS,idC}^{i}\}_{i \in [\eta]}, \eta, u, pms)$. This includes the states of all OWF-chains. Meanwhile, the client stores a variable u, which is initialised to be 1, to track the index of the stored PRG seed and the OWF-chain head.

In a proof generation process (corresponding to steps 220, 222 and 224 of the process 200B of Figure 2), to get a proof for time T, the client idC first determines the index i of the OWF-chain that should be used. The client idC then obtains the i-th OWF-chain head based on the current stored PRG seed ss_{idC}^{i} by literally running $u := u + 1$ and recursively computing $ss_{idC}^{u} || \rho_{idC}^{u} := G.Gen(ss_{idC}^{u-1})$ until $u = i$. Then, client idC can use the proof generation algorithm of Π_{OWF} (Figure 4B) to obtain the corresponding proof.

In a proof verification process (corresponding to step 226 of the process 200B of Figure 2), the proof verification algorithm of Π_{OWF} is used with the corresponding verification state $st_{idS,idC}^{i}$ idS,idC indicated by time T as input.

In some embodiments, a proof replenishment process may comprise the client idC running the algorithm $\Pi_{OWF}^{PRG}.Setup(1^{\kappa}, \rho_{idC}, T_{start}^{new}, \Delta_{rc}, T_{att}, \eta)$ for a new start time T_{start}^{new} to initialise a new PoA protocol instance with states (st_{idC}^{new}, $st_{idS,idC}^{new}$). To avoid needing to implement a typical digital signature scheme, which is too complex for a resource-constrained device, embodiments may use a one-time signature (OTS) method to securely distribute the state $st_{idS,idC}^{new}$. As there is only one state that needs to authenticate in each replenishment, a one-time signature (OTS) scheme is enough.

For example, Π_{OWF}^{PRG} can be modified to obtain a OTS scheme. The OTS scheme may be based on the OTS scheme introduced by Lamport (*Constructing digital signatures from a one-way function*, Technical Report CSL-98, SRI International, Palo Alto) (hereafter referred to as the Lamport scheme), for example.

To initiate a Lamport OTS scheme, the signer idC first samples a set of random values $ss_i \xleftarrow{\$} I_{OWF}$ for $i \in [2l_m]$ which form the signing key $sk_{idC} = \{ss_i\}_{i \in [2l_m]}$, where $l_m \in \mathbb{N}$ is the bit-length of the message to be signed. Then idC computes the public key $pk_{idC} = \{vk_i\}_{i \in [2l_m]}$, where $vk_i := F.Eval(ss_i)$. The public key is sent and stored (for example, in database 132) at the verifier idS. To sign a message m, idC splits m into l_m bits $m[1], m[2], \ldots, m[l_m]$, and the signature σ_{idC} for m is the set of secrets $\{ss_{2i-m[i]}\}_{i \in [l_m]}$. The verifier idS accepts (m, σ_{idC}) if it holds that $vk_{2i-m[i]} := F.Eval(ss_{2i-m[i]})$ for all indices $i \in [l_m]$, and rejects otherwise. Meanwhile, one can use any collision resistant hash function with the form CRH : $\{0,1\}^{*} \rightarrow \{0,1\}^{l_m}$ to compress a large message.

To modify Π_{OWF}^{PRG} to obtain a variant of the Lamport OTS scheme, first of all, the last $2l_m$ head OWF-chain nodes $x_0^{\eta-2l_m}, x_0^{\eta-2l_m+1}, \ldots, x_0^{\eta}$ can be reserved as the signing key, i.e., without sending them as the proofs. Note that the verify-point of the j-th OWF-chain for $j \in \{\eta - 2l_m, \eta - 2l_m + 1, \ldots, \eta\}$ can be seen as one of the verification keys. The signing procedure is identical to that of the Lamport OTS scheme. However, the signature verification procedure is changed as follows. The verifier idS accepts (m, σ_{idC}) if it holds that $\Pi_{OWF}.Verify\left(st_{idS,idC}^{j}, x_j, T_j\right) = 1$ for all indices $j \in \{\eta - 2l_m, \eta - 2l_m + 1, \ldots, \eta\}$, and rejects otherwise.

Eventually, to switch to a new protocol instance of Π_{OWF}^{PRG} (i.e., replenishing the proofs), idC could compute a hash value $m \coloneqq CRH(st_{idS,idC}^{new})$ and compute the signature σ_{idC} of m via the above modified Lamport OTS scheme. The signature σ_{idC} can be sent together with the subsequent new proofs. To avoid potential message loss errors, idC can replay σ_{idC} many times if necessary. In the meantime, idC only needs to store the PRG seed generated in the current time slot, and the PRG seed $ss_{idC}^{\eta-2l_m-1}$ for the 'signing key' generation. The number of signature-replayed-times is determined by the aliveness tolerance time T_{att}.

In Π_{OWF}^{PRG}, a PRG is leveraged to provide a separate channel for the client to quickly obtain the secret near the current time. Note that it is possible to properly instantiate the PRG so that the i-th seed can be quickly obtained from the $(i-1)$-th seed without calling G.Gen. For example, while instantiating the PRG with counter-mode based symmetric key encryption, the difference between the i-th and $(i-1)$-th seeds is only the 'counter'. Also, one can easily generalise the two-layer structure of Π_{OWF}^{PRG} into a multi-layer structure. However, this is a trade-off between fast proof generation and fast skipping proof generation. This is because if a multi-layer structure is used, one can quickly jump over a few sub-chains and compute the proof on a new subchain. In a PoA application, every single proof on the chains will be used in the protocol. More layers imply more PRG invocations, which will slow down the overall computation of the proofs. As will be shown later, in one example instantiation of PRG (i.e., CTR mode AES), it is possible to jump to any sub-chain via only one PRG operation. Accordingly, in at least some embodiments, the two-layer structure is necessary for proof auto-replenishment and sufficient for PoA application.

The number of sub-chains is lower-bounded by the number of secret keys in the Lamport OTS scheme. It may be beneficial to use the lower bound as the optimal number of sub-chains, since more sub-chains imply more PRG computations, which will be added to the overall performance overhead in setup and proof generation on the client side. The verification time is independent of the number of sub-chains, since the verifier can find

the index of a specific verification key based on the current time, and thus no PRG evaluation is needed.

An example architecture of a client device in which embodiments of the present disclosure may be implemented is shown in Figure 9. In some embodiments, each of the client devices 102.1, ..., 102.N shown in Figure 1 may be realised by the client device 800 of Figure 9.

As shown, the client device 800 includes the following components in electronic communication via a bus 806:

a) non-volatile (non-transitory) memory 804;
b) random access memory ("RAM") 808;
c) N processing components 810; and
d) a transceiver component 812 that includes N transceivers.

For example, the client device 800 may be a component, such as a programmable logic controller, remote terminal unit, or substation gateway, of an IIoT system such as an industrial control system, and the client device 800 may perform operations in a proof of aliveness protocol.

In other embodiments, the client device 800 may be a consumer electronic device such as a smart phone or tablet computing device, and the client device 800 may be provided with the ability to generate TOTPs, for example for authentication purposes for online services. In such embodiments, it will appreciated that the client device may comprise other components that are not illustrated in Figure 9, such as a display, user controls, a secure element, a NFC controller, and the like.

Although the components depicted in Figure 9 represent physical components, Figure 9 is not intended to be a hardware diagram. Thus, many of the components depicted in Figure 9 may be realised by common constructs or distributed among additional physical components. Moreover, it is certainly contemplated that other existing and yet-to-be developed physical components and architectures may be utilised to implement the functional components described with reference to Figure 9.

In general, the non-volatile data storage 804 (also referred to as non-volatile memory) functions to store (e.g., persistently store) data and executable code.

In some embodiments for example, the non-volatile memory 804 includes bootloader code, modem software, operating system code, file system code, and code to facilitate the implementation components, known to those of ordinary skill in the art, which are not depicted nor described for simplicity. For example, the non-volatile memory 804 may contain a chain generation module 822 and a proof generation module 824. The chain

generation module 822 may perform some or all of the operations shown being performed by client device 102.1 in process 200A. The proof generation module 824 may perform some or all of the operations shown being performed by client device 102.1 in process 200B.

In many implementations, the non-volatile memory 804 is realised by flash memory (e.g., NAND or ONENAND memory), but it is certainly contemplated that other memory types may be utilised as well. Although it may be possible to execute the code from the non-volatile memory 804, the executable code in the non-volatile memory 804 is typically loaded into RAM 808 and executed by one or more of the N processing components 810.

The N processing components 810 in connection with RAM 808 generally operate to execute the instructions stored in non-volatile memory 804. As one of ordinarily skill in the art will appreciate, the N processing components 810 may include a video processor, modem processor, DSP, graphics processing unit (GPU), and other processing components.

The transceiver component 812 includes N transceiver chains, which may be used for communicating with external devices via wireless networks. Each of the N transceiver chains may represent a transceiver associated with a particular communication scheme. For example, each transceiver may correspond to protocols that are specific to local area networks, cellular networks (e.g., a CDMA network, a GPRS network, a UMTS network), and other types of communication networks.

It should be recognised that Figure 9 is merely exemplary and in one or more exemplary embodiments, the functions described herein may be implemented in hardware, software, firmware, or any combination thereof. If implemented in software, the functions may be transmitted or stored as one or more instructions or code encoded on a non-transitory computer-readable medium 804. Non-transitory computer-readable medium 804 includes both computer storage medium and communication medium including any medium that facilitates transfer of a computer program from one place to another. A storage medium may be any available medium that can be accessed by a computer.

Figure 10 shows an example computing device 1000 that is capable of implementing verification server 110 of the system 100. In some embodiments, multiple computing devices 1000 may be considered to be a single verification server.

The components of the computing device 1000 can be configured in a variety of ways. The components can be implemented entirely by software to be executed on standard computer server hardware, which may comprise one hardware unit or different computer hardware units distributed over various locations, which may communicate over a

network. Some of the components or parts thereof may also be implemented by application specific integrated circuits (ASICs) or field programmable gate arrays.

In the example shown in Figure 10, the computing device 1000 is a commercially available server computer system based on a 32 bit or a 64 bit Intel architecture, and the processes and/or methods executed or performed by the computing device 1000 are implemented in the form of programming instructions of one or more software components or modules 1022 stored on non-volatile (e.g., hard disk) computer-readable storage 1024 associated with the computing device 1000. At least parts of the software modules 1022 could alternatively be implemented as one or more dedicated hardware components, such as application-specific integrated circuits (ASICs) and/or field programmable gate arrays (FPGAs).

The computing device 1000 includes at least one or more of the following standard, commercially available, computer components, all interconnected by a bus 1035:

(a) random access memory (RAM) 1026;
(b) at least one computer processor 1028, and
(c) a network interface connector (NIC) 1030 which connects the computer device 1000 to a data communications network (such as the public Internet and/or a local area network) and/or to external devices.

The computing device 1000 includes a plurality of standard software modules, including:

(a) an operating system (OS) 1036 (e.g., Linux or Microsoft Windows); and
(b) structured query language (SQL) modules 1042 (e.g., MySQL, available from http://www.mysql.com), which allow data to be stored in and retrieved/accessed from an SQL database 132.

The database 132 may form part of the computer readable data storage 1024. Alternatively, the database 132 is located remote from the computing device 1000 shown in Figure 10.

The boundaries between the modules and components in the software modules 1022 are exemplary, and alternative embodiments may merge modules or impose an alternative decomposition of functionality of modules. For example, the modules discussed herein may be decomposed into submodules to be executed as multiple computer processes, and, optionally, on multiple computers. Moreover, alternative embodiments may combine multiple instances of a particular module or submodule. Furthermore, the operations may be combined or the functionality of the operations may be distributed in additional operations in accordance with the invention. Alternatively, such actions may be embodied in the structure of circuitry that implements such functionality, such as the micro-code of

a complex instruction set computer (CISC), firmware programmed into programmable or erasable/programmable devices, the configuration of a field-programmable gate array (FPGA), the design of a gate array or full-custom application-specific integrated circuit (ASIC), or the like.

Each of the blocks of the flow diagrams of the processes performed by the verification server 110 may be executed by a module (of software modules 1022) or a portion of a module. The processes may be embodied in a non-transient machine-readable and/or computer-readable medium for configuring a computer system to execute the method. The software modules may be stored within and/or transmitted to a computer system memory to configure the computer system to perform the functions of the module.

The computing device 1000 normally processes information according to a program (a list of internally stored instructions such as a particular application program and/or an operating system) and produces resultant output information via input/output (I/O) devices 1030. A computer process typically includes an executing (running) program or portion of a program, current program values and state information, and the resources used by the operating system to manage the execution of the process. A parent process may spawn other, child processes to help perform the overall functionality of the parent process. Because the parent process specifically spawns the child processes to perform a portion of the overall functionality of the parent process, the functions performed by child processes (and grandchild processes, etc.) may sometimes be described as being performed by the parent process.

Example and performance characteristics

To show the practicality of the proposed PoA protocols Π_{OWF} and Π_{OWF}^{PRG}, they were implemented on embedded devices, and their performance evaluated on Raspberry Pi 3. The performance was evaluated in a single client scenario, instead of a multiclient scenario. This is because the multiple client setting is significant for security analysis, but the secrets and the chains of clients are independent of one another, and every client will operate independently. In other words, the performance of individual clients will remain the same in a multi-client setting.

In the example implementation, the one-way function (OWF) and pseudo-random generator (PRG) were instantiated with primitives in the random oracle model and the standard model, respectively. In the standard model instantiation, we implemented the OWF in Impagliazzo and Naor (referenced above), which is based on the subset sum problem, while the PRG was instantiated with the one built from OWF (Yu et al., "Pseudorandom number generators from regular one-way functions: New constructions with improved parameters", In *International Conference on the Theory and Application of*

Cryptology and Information Security, Springer, 261-279). At the core of the PRG, we still use the OWF based on the subset sum problem. In the random oracle model, counter-mode AES and SHA-2 family were used to instantiate the PRG and OWF respectively. The implementations of AES and SHA-2 are from the highly optimized MIRACL library. The system setup, proof generation, and proof replenishment time were measured on Raspberry Pi 3, and the proof verification time was measured on a laptop with Intel Core i7-8750H.

System Setup Time. We first measured the setup time of our PoA protocols, i.e., the time consumed for generating the verify-point to be shared with a verifier at the end of the setup procedure. The same amount of proofs (222) were initialised in Π_{OWF} and Π_{OWF}^{PRG} in both the random oracle model (Π_{OWF} RO) and the standard model (Π_{OWF} STD). Clearly, Π_{OWF} RO has the best performance in its setup, as all the primitives in the random oracle model are usually more efficient in computation than the ones in the standard model in practice. Also, when the performance of Π_{OWF} RO is compared with that of Π_{OWF}^{PRG} RO, Π_{OWF} RO is still faster, because, to generate the same amount of proofs, Π_{PRG}^{OWF} RO needs η additional PRG invocations. However, since Π_{OWF}^{PRG} has the auto-replenishment feature, we can instantiate a shorter chain, and replenish passwords when they are running out. In the implementations of Π_{OWF}^{PRG} RO* and Π_{OWF}^{PRG} STD*, shorter chains are set up initially, so they lead to a better setup performance than Π_{OWF} RO.

Proof Generation Time. As a proof of aliveness scheme, the prover has to send every single node on the chain to the verifier. Also, typically, PoA is running on an embedded device which has very constrained computation power, so we have to wisely schedule its computation. The best strategy will be to divide the whole chain into multiple segments with the same length and save the head of each segment as a checkpoint in a non-volatile memory. Moreover, one can store the last segment after the setup process in a non-volatile memory as well. After the PoA starts, the client sends one proof in one segment in the reverse order and generates one proof in the previous segment in the forward order. In this way, the client always stores one full segment of nodes in the memory. Suppose that in a time slot, the corresponding proof is the j-th node from the end of segment k (counting from the head of the chain), then the proof will be fetched directly from the memory and be sent to the verifier. In the same time slot, one new proof (the j-th node from the head of segment $(k-1)$ will be generated and replace the node used in the current time slot, such that when the nodes in segment k are used up, all the nodes in segment $(k-1)$ will be ready to use. Using this strategy, we equally distribute the proof generation time into every time slot and optimize its performance in the worst case, because in every time slot the client only needs to fetch one node from the storage and generate one new node. Thus, it only requires a one-way function evaluation in every time

slot for Π_{OWF} in both the average case and the worst case. An additional pseudorandom number generator evaluation is needed in the worst case of Π_{OWF}^{PRG}. Apparently, this strategy requires memory space, and the most memory-efficient checkpointing strategy will be to use $\sqrt{N}$ checkpoints, which results in a memory requirement of $2\sqrt{N}$ nodes ($\sqrt{N}$ checkpoints, and $\sqrt{N}$ nodes per segment). Moreover, if one can shorten the overall length of the chain N, PoA can achieve better setup performance and save memory space as well. Fortunately, our Π_{OWF}^{PRG} allows replenishment of proofs, so we can easily shorten the length of the chain and achieve better performance as shown in the cases of Π_{OWF}^{PRG} RO* and Π_{OWF}^{PRG} STD*.

Proof Verification Time. The proof verification time, as shown in Table 1 below, is the execution time of one OWF on the server for all protocols. In case of network failures and packet loss, the verification can take as many as T_{att}/Δ_s times of the OWF evaluation. In practice, for example, when T_{att} is 5 minutes, and Δ_s is 30 seconds, then the verifier needs to run at most 300/30 = 10 times of OWF to verify the proof, and the verification time of protocols instantiated in the random oracle model and the standard model can be up to 4.7 μs and 41.2 μs respectively.

Table 1: summary of evaluation results

Protocol	Setup	Generation	Ver	Rep
Π_{OWF} RO	15.69 s	3.74 μs/ 3.74 μs	0.47 μs	N/A
Π_{OWF} STD	185.33 s	44.19 μs / 44.19 μs	4.12 μs	N/A
Π_{OWF}^{PRG} RO	17.11 s	4.07 μs / 18.00 μs	0.47μs	2.65 ms
Π_{OWF}^{PRG} RO*	0.17 s	5.50 μs / 18.00 μs	0.47μs	2.65 ms
Π_{OWF}^{PRG} STD	192.48 s	45.5 μs / 10.46 ms	4.12 μs	5.28 s
Π_{OWF}^{PRG} STD*	6.81 s	205.49 μs / 10.46 ms	4.12 μs	5.28 s

(i) The parameters of each protocol are as follows: $\ell = 2^{15}$; assuming there are $\sqrt{N}$ checkpoints for PoA. (1) Π_{OWF} RO: $N = 2^{22}$; (2) Π_{OWF} STD: $N = 2^{22}$, $l_r = 448$; (3) Π_{OWF}^{PRG} RO: $N = 2^{22}$, $\eta = 512$; (4) Π_{OWF}^{PRG} RO*: $N = 2^{15}$, $\eta = 512$; (5) Π_{OWF}^{PRG} STD: $N = 2^{22}$, $\eta = 512$, $l_r = 430$; (6) Π_{OWF}^{PRG} STD*: $N = 2^{15}$, $\eta = 512$, $l_r = 430$, ℓ is the number of clients (which would affect the length of l_r according to the security analysis), N is the number of proofs, l_r is the input size of OWF, and η is the number of sub-chains in Π_{OWF}^{PRG}.

(ii) The proof generation time in average case and the worst case are reported separately in the form of average/worst. "Ver" and "Rep" stand for "Verification" and "Replenishment".

(iii) The reported proof replenishment time does not include the setup time of a new set of proofs.

Proof Replenishment Time. When the proofs are about to be used up, Π_{OWF}^{PRG} RO and Π_{OWF}^{PRG} STD will start to generate the next chain and create the one-time signature for it. This is very important for PoA protocols because critical infrastructure cannot be stopped to replenish new proofs, and there is no way to securely commit the new proof over an insecure communication channel. We measured the time consumed for constructing an OTS according to the hash value of a new verify-point, and the performance is reported in Table 1.

Parameter Selection in Practice. The concrete parameter setting typically depends on the specific application and the specific system in which the method is to be deployed. The general methodology is to first identify the time-to-critical-states of the system. Then we can use the minimal time-to-critical-states as the aliveness tolerant time T_{att}, or more conservatively, multiply a certain constant (e.g., 0.5) with it. The selection of the period of a time slot Δ_s needs to guarantee that at least one proof will be delivered to the server within T_{att}, considering the benign packet loss and network failure. The life-span of a PoA instance Δ_{rc} can be the time between two planned maintenances of the system, as the system has to stop during maintenance.

Case Study for Parameter Selection. Let us take a real-world water treatment testbed at Singapore University of Technology and Design (SWaT) as an example. Researchers have shown that the minimal time-to-critical-states of the first stage (raw water processing) of SWaT is 149 seconds (2 minutes and 29 seconds). Thus 149 seconds can be set as T_{att} for this stage, and Δ_s can be set to 30 seconds. Given the parameters above, N = 222 will be able to support the system to operate continuously for 4 years, and therefore Δ_{rc} can be around 4 years. Also, Π_{OWF}^{PRG} offers proof replenishment, and effectively Δ_{rc} will not be limited by N anymore.

As can be seen from the above results, the proof generation time is in the order of microseconds, so the methods disclosed herein are sufficiently efficient to be deployed in real-world devices.

Methods, devices and systems according to the present disclosure can be used in any systems for which maximised uptime is desired.

For example, at the core of critical infrastructure, usually Programmable Logic Controllers (PLCs) are used to control physical processes directly. It is crucial that the PLCs are running all the time and are actually controlling the processes. In 2007, a power system in California was shut down by a disgruntled contractor by shutting off the power of some computers. If an attacker is more sophisticated and able to inject packets into the network traffic to deceive the control centre, the power systems will not be able to respond to this incident immediately. Proof of aliveness can be integrated into the firmware of PLCs to

detect any interrupted executions of the control logic. Note that on PLCs, most attacks occur on the control logic layer, which is one layer above the firmware.

In another example, proof of aliveness may be implemented in implantable medical devices such as pacemakers. A proof of aliveness protocol as disclosed herein can be easily integrated into an implantable device to keep it connected with an external device such as a server maintained by a healthcare provider (such as the patient's specialist), a patient's smartphone, or other smart wearable devices.

In a further example, to prevent replay attacks that are used to compromise surveillance cameras, a PoA protocol as disclosed herein may be implemented inside surveillance cameras to embed a proof into every frame as an invisible watermark, such that the administrators can automatically verify the freshness of each frame.

Many modifications will be apparent to those skilled in the art without departing from the scope of the present invention.

Throughout this specification, unless the context requires otherwise, the word "comprise", and variations such as "comprises" and "comprising", will be understood to imply the inclusion of a stated integer or step or group of integers or steps but not the exclusion of any other integer or step or group of integers or steps.

The reference in this specification to any prior publication (or information derived from it), or to any matter which is known, is not, and should not be taken as an acknowledgment or admission or any form of suggestion that that prior publication (or information derived from it) or known matter forms part of the common general knowledge in the field of endeavour to which this specification relates.

CLAIMS

1. A method for generation and/or authentication of time-based one-time passwords (TOTPs), comprising, at a client device:

   performing an initialisation operation comprising:

   determining a time period;

   generating one or more one-way function (OWF) chains for each of one or more sub-periods of the time period, wherein an OWF chain for a sub-period is generated by:

   dividing the sub-period into N segments of equal duration; and

   recursively applying a one-way function N times to a random initial secret that is a head node of the OWF chain, to generate a tail node of the OWF chain; and

   transmitting, to one or more verifier devices, an initialisation message comprising a client identifier and the tail nodes of respective OWF chains, such that a TOTP generated by the client device is verifiable by the one or more verifier devices using one of the tail nodes as a verify point; and

   performing a proof generation operation comprising:

   determining a number M of segments remaining from a current time until the end of a current sub-period of the one or more sub-periods;

   generating a TOTP by M recursive evaluations of the OWF, starting with a head node of the OWF chain for the current sub-period; and

   transmitting an authentication message comprising the client identifier and the TOTP to one of the verifier devices.

.

2. A method according to claim 1, wherein a head node of an OWF chain for each sub-period is generated by a pseudo-random number generator using a secret seed for the immediately preceding sub-period.

3. A method according to claim 2, further comprising an auto-replenishment operation prior to all nodes of the OWF chain or OWF chains being consumed.

4. A method according to claim 3, wherein the auto-replenishment operation comprises distributing one or more additional verify points to the one or more verifier devices using a one-time signature scheme.

5. A method according to any one of claims 1 to 4, comprising, at a verifier device of the verifier devices, performing an authentication operation comprising:

receiving the client identifier and the TOTP;

determining, based on a time of receipt of the authentication message, a number Z of time segments that have elapsed since a time of last authentication of the client device;

generating verification data by Z recursive evaluations of the OWF, starting with the TOTP; and

if the verification data matches the verify point for a current sub-period, updating the verify point to be the TOTP; and updating the time of last authentication to the time of receipt of the authentication message.

6. A method according to any one of claims 1 to 5, wherein the proof generation operation is performed at regular intervals as a proof of aliveness of the client device.

7. A method according to claim 6, wherein the authentication operation further comprises determining whether a difference between the time of receipt of the authentication message and the time of last authentication of the client device exceeds an aliveness time threshold.

8. An apparatus for generation of time-based one-time passwords (TOTPs), comprising at least one processor in communication with computer-readable storage, the computer-readable storage having instructions stored thereon for causing the at least one processor to:

perform an initialisation operation comprising:

determining a time period;

generating one or more one-way function (OWF) chains for each of one or more sub-periods of the time period, wherein an OWF chain for a sub-period is generated by:

dividing the sub-period into N segments of equal duration; and

recursively applying a one-way function N times to a random initial secret that is a head node of the OWF chain, to generate a tail node of the OWF chain; and

transmitting, to one or more verifier devices, an initialisation message comprising a client identifier and the tail nodes of respective OWF chains, such that a TOTP generated by the client device is verifiable by the one or more verifier devices using one of the tail nodes as a verify point; and

to perform a proof generation operation comprising:

determining a number M of segments remaining from a current time until the end of a current sub-period of the one or more sub-periods;

generating a TOTP by M recursive evaluations of the OWF, starting with a head node of the OWF chain for the current sub-period; and

transmitting an authentication message comprising the client identifier and the TOTP to one of the verifier devices.

.

9. An apparatus according to claim 8, wherein the instructions further cause a secret seed to be generated for each sub-period by a pseudo-random number generator using a head node of an OWF chain for the immediately preceding sub-period.

10. An apparatus according to claim 9, wherein the instructions further cause the at least one processor to perform an auto-replenishment operation prior to all nodes of the OWF chain or OWF chains being consumed.

11. An apparatus according to claim 10, wherein the auto-replenishment operation comprises distributing one or more additional verify points to the one or more verifier devices using a one-time signature scheme.

12. An apparatus according to any one of claims 8 to 11, wherein the proof generation operation is performed at regular intervals as a proof of aliveness of the apparatus.

13. A verification server comprising at least one processor in communication with computer-readable storage, the computer-readable storage having stored thereon instructions for causing the at least one processor to perform an authentication operation comprising:

receiving a client identifier and a TOTP from a client device;

retrieving a stored verify point and a time of last authentication for the client device based on the client identifier and a current time;

retrieving a one-way function (OWF);

determining, based on a time of receipt of the authentication message, a number Z of time segments that have elapsed since the time of last authentication;

generating verification data by Z recursive evaluations of the OWF, starting with the TOTP; and

if the verification data matches the verify point for the current time, updating the verify point to be the TOTP; and updating the time of last authentication to the time of receipt of the authentication message.

14. A verification server according to claim 13, wherein the authentication operation further comprises determining whether a difference between the time of receipt of the authentication message and the time of last authentication of the client device exceeds an aliveness time threshold.