Proof of Persistent Aliveness

Xuelian Cao, Zheng Yang, Jianting Ning, Chenglu Jin, Zhiming Liu and Jianying Zhou

***Abstract*—Proof of Aliveness (PoA) has emerged as a useful cryptographic concept for periodically ascertaining the operational status (aliveness) of devices, especially for those in cyber-physical systems. However, existing PoA schemes exhibit shortcomings stemming from intermittent aliveness proofs and a lack of resilience against the threats caused by malicious verifiers. Motivated by this, we introduce a new security notion called *Proof of Persistent Aliveness* (PoPA), which encompasses two new properties: persistent aliveness (PAlive) and audit (Audit). Our PAlive strengthens prior work by addressing the security concerns associated with generating persistent aliveness proofs in a continuous time manner, while Audit covers the threats posed by malicious verifiers. To efficiently realize PoPA, we developed two new building blocks: a deterministic hash-based Proof of Work (HPoW) scheme and private tweakable hash (PTH) functions. Using these primitives, we propose a scalable and lightweight PoPA construction, named SPAC, which is provably secure in our PoPA model without relying on random oracles. SPAC leverages HPoW and a customized authenticated credential structure that employs a variant of the Winternitz one-time signature scheme derived from PTH, enabling unlimited aliveness proofs with very small proof size. Over 93% of aliveness proofs are 84 bytes in size, with the worst-case proof size being only 372 bytes.**

***Index Terms*—Proof of Aliveness, Persistent Aliveness, Proof of Work, Lightweight Authentication, Security Model**

I. Introduction

In CPS, "aliveness" typically refers to ensuring that components or entities within the system are operational and active. Aliveness (and aliveness detection) is also widely used in network operations, administration, and maintenance for the control plane of Internet Protocol and Multiprotocol Label Switching networks. For example, the Internet Engineering Task Force's standard, *Path Computation Element Communication Protocol Generic Requirements* [1], specifies that communication between path computation clients must include session *aliveness detection* and recovery so that both sides can rapidly detect failures.

Proof of Aliveness (PoA) [2], [3] is a lightweight cryptographic concept designed for cyber-physical systems (CPS). It enables a client device to non-interactively demonstrate its aliveness to a control center. The purpose of PoA is to attest the continuous and reliable operation of critical components, particularly in applications where uptime and performance are crucial. A recent report [4] on the Baltimore bridge collapse highlights the importance of CPS devices' aliveness, as the failure of the cargo ship Dali's critical systems led to the vessel drifting and colliding with the Key bridge, causing the span to fall into the river and resulting in six deaths. Cyber attackers disrupting power grids can plunge hundreds of thousands of people into darkness for extended periods [5]. Such denial-of-service (DoS) attacks have immediate repercussions on users and customers, while other malicious actors may surreptitiously disable specific services to evade detection. These incidents collectively underscore the need for PoA mechanisms that provide authenticated, time-bounded aliveness evidence under adversarial conditions and give operators confidence that critical infrastructure remains operational and secure. From an engineering perspective, PoA can run as a low-priority background service, co-scheduled with primary tasks using standard controls (priority settings, CPU shares/cgroups, core pinning, and safe scheduling points). Additionally, PoA's configurable parameters (e.g., check window length and duty cycle) can make the resource footprint explicit and enable integration without materially affecting functional operability. In practice, PoA is used in applications like Singapore SMRT metro system [6], where the Evacuation Control system receives a heartbeat signal from Automatic Train Control every 300 milliseconds. If the signal is not received within 10 seconds, the evacuation protocol is triggered, indicating a train is unaware of other trains or track conditions. However, such a PoA scheme is vulnerable to attackers who can forge aliveness messages. The system model and main threats of PoA (PoPA) are depicted in Figure 1 for a high-level understanding.

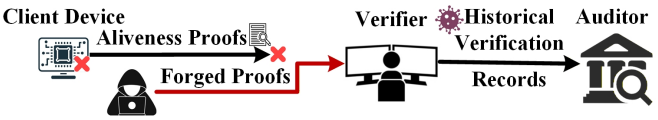


Fig. 1: Illustration of System Model and Threats.

To achieve provably secure PoA, Jin et al. [2], [3] proposed two PoA constructions utilizing time-based one-time passwords (TOTP). The key design principle is to emit one-time passwords periodically (e.g., every 30 seconds) as aliveness

Zheng Yang and Jianting Ning are the corresponding authors.

X. Cao, Z.Yang and Z. Liu are with Southwest University, No.2 Tiansheng Road Beibei District, Chongqing 400715, P.R.China; Emails: xuelian-cao7@email.swu.edu.cn and {youngzheng,zhimingliu88}@swu.edu.cn

J. Ning is with the School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, P.R.China; and with the Faculty of Data Science, City University of Macau, Macau 999078, P.R.China; and also with the Key Laboratory of Analytical Mathematics and Applications of Fujian Normal University (Ministry of Education), Cangshan District, Fuzhou 350007 P.R. China. E-mail: jtning88@gmail.com

C. Jin is with CWI Amsterdam, Science Park 123 1098 XG Amsterdam, Netherlands. E-mail: chenglu.jin@cwi.nl

J. Zhou is with the iTrust, Singapore University of Technology and Design, 8 Somapah Rd, Singapore, 487372. E-mail: jianying_zhou@sutd.edu.sg

This paper has supplementary downloadable material available at http://ieeexplore.ieee.org., provided by the author. The material includes main notations, a DGTOTP variance with cache, proofs of Theorem 1 and Theorem 2, and security enhancement for ACACCE. Contact youngzheng@swu.edu.cn for further questions about this work.

This work was supported by the Natural Science Foundation of China under Grant Grant No. 12441101 and No. 62372386, the National Cryptography Science Foundation of China under Grant No. 2025NCSF02055, and by the National Key Research and Development Program of China under Grant No. 2025YFE0220300.

proofs, mimicking heartbeat signals from client devices.[1] These schemes have been implemented in a cryptographic library [7] for Programmable Logic Controllers (PLCs), demonstrating their practicality in real-world applications. Jin et al.'s schemes also exhibit fault-tolerance and allow some benign loss of aliveness proofs, as long as the verifier receives at least one proof within a specified aliveness tolerance time $\tilde{T}_a$ (e.g., $\tilde{T}_a = 3$ minutes). However, they only prove aliveness intermittently, leaving room for attackers to disrupt aliveness within the tolerance window, such as by temporarily shutting down and rebooting clients to avoid detection. This intermittent nature renders them insufficient to guarantee the continuous and uninterrupted operation of critical devices. Therefore, it is crucial to ensure the persistent aliveness of devices in certain practical scenarios, e.g., power grids or autonomous vehicles.

Motivations. To the best of our knowledge, the challenge of *generating persistent aliveness proofs*, essential for maintaining a client's operational status with minimal disruption, remains unsolved. Moreover, verifiers themselves could become prime targets for attackers seeking intrusion avenues (e.g., exploiting vulnerabilities in the verifier's system, such as Solarwinds hack [8], [9]), aiming to disable cybersecurity monitoring and cease responding to client death incidents. While the compromise of verifiers may be inevitable at times, we need to prevent malicious verifiers from manipulating aliveness proofs, e.g., tampering with aliveness proofs to hide the death events of clients. This requires a new security feature that can audit a verifier's historical verification state to identify whether it fails to fulfill its duty of verifying aliveness proofs and carrying out accident management. Meeting this audit requirement is challenging without costly methods like blockchain. Instead, we aim to achieve auditability through the PoA scheme's design, suited for CPS environments where clients have insecure connections to a verifier. Notably, Jin et al.'s chain-based TOTP schemes and their variants [3] lack the audit property, as a current received password can derive previously undelivered passwords, concealing the fact of its failure to complete the verification work. Thus, *formulating audit and finding provably secure solutions satisfying it*, remain open challenges.

Our Work. We address the aforementioned open questions by developing the security notion of Proof of Persistent Aliveness (PoPA), building upon prior work [2]. Our PoPA security model focuses on two key security properties: persistent aliveness (PAlive) and audit (Audit). We adapt PAlive from [2] (in which it uses intermittent time slots) to consider the running time of an algorithm via the number of elementary operations run on a Turning Machine, enabling us to formulate the continuous proof generation processes. Specifically, we further define the PAlive property through two kinds of adversaries: a *forgery-adversary* being capable of forging aliveness proofs from uncorrupted clients, and a *time-breaker* simulating the ability to generate proofs much faster than legitimate procedures. Moreover, we define the Audit property to address malicious verifiers. A secure PoPA with Audit should guarantee that no adversaries can present malicious verification states involving verification records prior to time T^*, even if the corresponding client is corrupted after T^*.

Furthermore, we propose a lightweight PoPA construction, named SPAC, which can be proven secure under our defined security model without random oracles. The key challenge in designing a lightweight PoPA scheme is avoiding public-key cryptographic (PKC) techniques to make it suitable for resource-constrained devices in IoT and CPS[2], while maintaining post-quantum security, thus excluding PKC-based primitives like verifiable delay functions [10] and time-lock puzzles [11]. In SPAC, we first leverage a hash-based proof of work (HPoW) scheme to represent a client's effort within a set aliveness-time threshold. The intuition behind this is that only an alive client can promptly solve the HPoW puzzle. Besides, HPoW is efficient for resource-constrained devices and easy to embed authentication credential for Audit. Unlike naively using existing HPoW schemes, we customize a deterministic HPoW scheme based on truncation collision-resistant (TrCR) hash functions [12], preventing adversaries from manipulating HPoW solutions, instead of relying on random nonces [13].

To efficiently demonstrate that the HPoW solution is from the corresponding client, we build a forward-secure one-time authenticated credential structure (ACS) that continuously produces one-time credentials for generating HPoW puzzles. Only the client holding the unreleased credentials can solve the puzzle, allowing us to set arbitrary HPoW difficulty to meet proof-generation time requirements for resource-constrained devices and applications. A key feature of ACS is its ability to self-replenish credentials efficiently, avoiding the use of costly forward-secure digital signatures (FSDS) [14], [15] for authentication. This overcomes limitations in existing replenishment schemes [2], [3] in achieving the Audit property. To build ACS, we adapt the chain-based one-time signature (OTS) scheme WOTS+C [16], creating a variant called WOTS+CP for credential replenishment. This variant combines each signature-chain of WOTS+C with a one-time credential chain, using a new private tweakable hash (PTH) function that leverages private tweaks (i.e., one-time credentials). Unlike WOTS+C's public tweaks, PTH ensures forward secrecy of credentials, enabling Audit of SPAC.

The final component of our ACS is using a Merkle tree (MT) to compress and authenticate the leaves which are linked to the tails of signature-chains in WOTS+CP via PTH. This allows us to initialize a new ACS instance and sign it gradually during the proof generation process rather than all at once. The process starts with the client using the preimage (including the credential for solving the HPoW puzzle and the last signing-key element in a signature-chain) of a Merkle tree leaf, and then proceeds with other credentials and signing-key elements in reverse order until reaching the corresponding signature value. Upon receiving the final signature value, the verifier can verify the signature. As a result, SPAC with the replenishment feature can enable unlimited aliveness proofs

[1] Specifically, they employ an N-node one-way function (OWF) chain, where each node $x_i = f(x_{i-1})$ is derived from a random seed x_0 and revealed in reverse. The verifier, holding the tail x_N, verifies each proof through iterative applications of the OWF.

[2] E.g., [7] shows that PKC-related operations are inefficient on PLC.

with a small proof size. The compact proof size minimizes network congestion, making it ideal for large-scale monitoring of critical infrastructure.

Contributions. We make the following contributions:

1) We advance the development of the cryptographic concept of PoA by introducing the security notion of PoPA, which includes two new properties: PAlive and Audit. Our PAlive is strengthened from prior work to tackle the security of aliveness proofs generated continuously over time, while Audit is a completely new security property for addressing threats posed by malicious verifiers.
2) To be independent of interest, we devise two new building blocks: a deterministic HPoW scheme and PTH functions, crucial for realizing PoPA efficiently. The customized HPoW scheme is compatible with a lightweight authentication scheme (without message authentication). Meanwhile, PTH is proposed to adapt WOTS+C to create a variant called WOTS+CP, guaranteeing forward secrecy of signing-key elements.
3) We propose SPAC, proven secure within our PoPA model without random oracles. SPAC showcases the utilization of HPoW and a customized ACS derived from WOTS+CP to realize PoPA, supporting unlimited aliveness proofs with highly compact proof size. The proof size of SPAC is particularly compact, with over 93% of proofs being just 84 bytes and a worst-case size of 372 bytes.

II. Related Work

Proof of Work. PoW, introduced by Dwork and Naor [17], is a cornerstone of cryptography, initially designed to mitigate DoS attacks and email spam [18] but now fundamental to modern cryptocurrencies [19]. Efforts to construct PoWs with provable security [20], [21] include Ball et al.'s [21] approach, leveraging worst-case hard problems like k-Orthogonal Vectors (k-OV). Ball et al.'s PoWs operate in time $\tilde{O}(n^k)$ for k-OV function evaluation, with the non-interactive version requiring random oracles and additional $O(k^2)$ hash operations. Yet, this complexity may be inefficient for constrained devices. Therefore, we reconsider the security and performance requirements of PoWs to better suit our PoPA constructions, aiming to develop a lightweight, non-interactive, deterministic, and standard-model secure hash-based PoW.

One-time Passwords. The first PoA scheme [2] and its optimized variant [3] are based on asymmetric time-based one-time passwords (TOTP), valued for their lightweight design and resistance to verifier compromise. Asymmetric TOTP schemes, such as S/KEY [22], T/Key [23], and the aforementioned PoA schemes, employ chain structures derived from one-way functions (OWFs), inspired by Lamport's work [24]. In Lamport-style single-chain TOTP frameworks, each password is computed by applying an OWF to the previous one, but these schemes lack forward secrecy, as the verifier can compute all prior passwords from the latest one. This limitation renders them unsuitable for constructing PoPA schemes with the audit property, highlighting the need for independent authentication credentials to generate aliveness proofs and enable audit.

Forward Secure Digital Signatures. FSDS [25] can protect past signatures even if the current key is compromised. This can be achieved through the periodic update and deletion of keys, significantly enhancing breach resilience in critical areas like secure audit logging and digital forensics. Hence, we can construct a naive PoPA with the Audit by leveraging FSDS to sign a PoW solution. However, this straightforward construction might be inefficient for resource-constrained devices, though various forward-secure digital signature schemes offer different trade-offs in performance (e.g., [14], [26]–[29]), balancing factors like signature and public key sizes, update frequency, and computational efficiency. In particular, as the message authentication feature is not mandatory in PoPA, FSDS-based PoPA schemes are computationally costly compared to our PoPA solutions utilizing only forward secure one-time authentication credentials.

Our PoPA scheme is compared with the naive FSDS solution in Section VI.

III. Preliminaries

General Notations. We denote the security parameter by κ, the string consisting of κ ones by 1^κ, the empty string by $\emptyset$, and the set of integers between 1 and n by $[n] = \{1, \ldots, n\} \subset \mathbb{N}$. For a binary string s, let $|s|$ be the bit-length and $s(i)$ denote the i-th bit of s. Let $\|$ be the string concatenation operation. For a finite set $\mathcal{X}$, we use $x \xleftarrow{\$} \mathcal{X}$ to denote that an element x is sampled from $\mathcal{X}$ uniformly at random. We assume the parameters of each primitive are implicitly used in its algorithms.

We provide security definitions for the following building blocks using the concrete security approach [30]. To simplify the security analysis significantly, we assume that the runtime of an adversary in the security definitions includes the time of the security experiment. Meanwhile, the running time of an algorithm is considered as the number of elementary operations run on a Turning Machine.

Proof of Work. A Proof of Work scheme $\mathsf{PoW} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Solve}, \mathsf{Verify})$ consists of the following algorithms. $\mathsf{Setup}(1^\kappa)$ takes as input 1^κ, and sets up a system parameter pms_{PoW} defining a difficulty space $\mathcal{D}_{\mathsf{PoW}}$, challenge-message space $\mathcal{M}_{\mathsf{PoW}}$, puzzle space $\mathcal{P}_{\mathsf{PoW}}$, and solution space $\mathcal{S}_{\mathsf{PoW}}$. The puzzle-generation algorithm $\mathsf{Gen}(d, cm)$ generates a puzzle $p \in \mathcal{P}_{\mathsf{PoW}}$ given a difficulty $d \in \mathcal{D}_{\mathsf{PoW}}$ and challenge-message $cm \in \mathcal{M}_{\mathsf{PoW}}$. $\mathsf{Solve}(p)$ takes as input a puzzle p, and produces a solution $s \in \mathcal{S}_{\mathsf{PoW}}$. The verification algorithm $\mathsf{Verify}(p, s)$ takes as input a puzzle $p \in \mathcal{P}_{\mathsf{PoW}}$, and a solution $s \in \mathcal{S}_{\mathsf{PoW}}$. It outputs 1 if s is a valid solution to p, and outputs 0 otherwise. For any $d \in \mathcal{D}_{\mathsf{PoW}}$ and any $cm \in \mathcal{M}_{\mathsf{PoW}}$, a specific PoW scheme P should satisfy the following properties: i) *Efficiency*: $\mathsf{P.Gen}(d, cm)$, $\mathsf{P.Verify}(p, s)$ are computable in time $\tilde{O}(T'(d))$, and $\mathsf{P.Solve}(p)$ is computable in time $\tilde{O}(T(d))$, where $T(d) \geq T'(d)$; ii) *Completeness*: for any $p \leftarrow \mathsf{P.Gen}(d, cm)$ and $s \leftarrow \mathsf{P.Solve}(p)$, it holds that $\mathsf{P.Verify}(p, s) = 1$; and iii) *Hardness*: no adversaries can solve the puzzle significantly faster than $T(d)$.

We adapt the security game $G^{\mathsf{HARD}}_{\mathcal{A},\mathsf{P}}(\kappa, d)$ from [20], [21] to formalize the hardness of a PoW scheme P, as shown in Figure 2.

Proc.Init() :	**Proc.PChallenge() :**
OUTPUT $\mathsf{P.Setup}(1^\kappa)$	$cm \xleftarrow{\$} \mathcal{M}_{\mathsf{PoW}}$, $p \leftarrow \mathsf{P.Gen}(d, cm)$, OUTPUT d, cm, p
Proc.Finalize(s^*) :	
IF $\mathsf{P.Verify}(p, s^*) = 1$, OUTPUT 1, ELSE OUTPUT 0	

Fig. 2: Procedures of the Security Game $G^{\mathsf{HARD}}_{\mathcal{A},\mathsf{P}}$.

Definition 1. *We say that a* PoW *scheme* P *is* $(d, T(d), \epsilon^d_{\mathsf{PoW}})$*-hard if for any polynomial-size adversary* $\mathcal{A}$ *running in time* $T(d)^\tau$ *with any constant* $\tau < 1$*, it holds that* $\Pr[G^{\mathsf{HARD}}_{\mathcal{A},\mathsf{P}}(\kappa, d) = 1] \leq \epsilon^d_{\mathsf{PoW}}$ *under a given difficulty* d*.*

Truncation Collision-resistant Hash Functions. We first define keyed hash (KH) functions $\mathsf{H} = (\mathsf{Setup}, \mathsf{Eval})$ with two algorithms. On input 1^κ, $\mathsf{Setup}(1^\kappa)$ outputs system parameters pms_{KH} defining the hash key space $\mathcal{K}_{\mathsf{KH}}$, message space $\mathcal{M}_{\mathsf{KH}}$, and hash value space $\mathcal{Y}_{\mathsf{KH}}$. The evaluation algorithm $\mathsf{Eval}(hk, m)$ takes a hash key $hk \in \mathcal{K}_{\mathsf{KH}}$ and message $m \in \mathcal{M}_{\mathsf{KH}}$, and produces a hash value $\alpha \in \mathcal{Y}_{\mathsf{KH}}$. We write $\mathsf{H}(m)$ for $\mathsf{H.Eval}(hk, m)$ when hk is clear from the context.

We denote with $\mathsf{H}[i]$ the evaluation of a keyed hash function H and the truncation of the corresponding output to the first i bits. The truncation collision resistance (TrCR) property ensures that no algorithm can achieve significantly better collision discovery rates than the standard birthday collision algorithm [12], even when accounting for hash value prefixes $\mathsf{H}[i](m)$ of $\mathsf{H}(m)$. It is straightforward to see that if the output of H is not truncated (i.e., $i = |\mathcal{Y}_{\mathsf{KH}}|$), then the above security definition implies the standard *collision resistance*. In addition, we may require H to satisfy a weaker security notion called *multi-function multi-target extended target-collision resistance* (MM-eTCR) [31].

We define the security game $G^{\mathsf{TrCR}}_{\mathcal{A},\mathsf{KH}}(\kappa, q, i)$ for the truncation collision-resistant hash H in Figure 3 with a specific truncation bit-length i. Let $q \in \mathbb{N}$ represent the adversary's parallel processing capability in a single step, as determined by the adversary.

Proc.Init() :	**Proc.Finalize($m_1, \ldots, m_q$) :**
$pms_{\mathsf{KH}} \leftarrow \mathsf{H.Setup}(1^\kappa)$	IF $\exists\ (u, v) \in [q]$ s.t. $\mathsf{H}[i](m_u) = \mathsf{H}[i](m_v) \bigwedge m_u \neq m_v$
$hk \xleftarrow{\$} \mathcal{K}_{\mathsf{KH}}$	OUTPUT 1
OUTPUT pms_{KH}, hk	OUTPUT 0

Fig. 3: Procedures of the Security Game $G^{\mathsf{TrCR}}_{\mathcal{A},\mathsf{KH}}$.

Definition 2. *We say that* H *is a* $(q, i, T, \epsilon^i_{\mathsf{TrCR}})$*-secure* TrCRH *family if it holds that* $\Pr[G^{\mathsf{TrCR}}_{\mathcal{A},\mathsf{KH}}(\kappa, q, i) = 1] \leq \epsilon^i_{\mathsf{TrCR}}$ *for any polynomial-size adversary* $\mathcal{A}$ *running in time* T*, where* $\epsilon^i_{\mathsf{TrCR}} \leq \frac{T(T-1)}{2^{i+1}}$.

Other Security Properties of Keyed Hash Functions. We provide the formal security definitions of required security properties for the KH functions H used in Section V-C. That is, we may require H to satisfy the multi-function multi-target extended target-collision resistance (MM-eTCR) [31], the single-function multi-target second-preimage resistance (SM-SPR), the one-wayness (SS-OW) and the undetectability (SS-UD) in the context of single-function single-target. The SM-SPR, SS-OW, and SS-UD are simplied from the distinct-function multi-target counterparts in [32]. The corresponding security games (in Figures 4, 5, 6, and 7, respectively) and definitions are presented as follows.

Proc.Init() :	**Proc.HChallenge(m) :**
$pms_{\mathsf{KH}} \leftarrow \mathsf{H.Setup}(1^\kappa)$	$cnt := cnt + 1$, $hk_{cnt} \xleftarrow{\$} \mathcal{K}_{\mathsf{KH}}$
$cnt := 0$	APPEND $(cnt, hk_{cnt}, m) \rightarrow$ HL
OUTPUT pms_{KH}	OUTPUT hk_{cnt}
Proc.Finalize(j, hk^*, m^*) :	
IF $\exists (j, hk_j, m_j) \in$ HL *s.t.* $\mathsf{H}(hk_j, m_j) = \mathsf{H}(hk^*, m^*)$ $\bigwedge (hk_j, m_j) \neq (hk^*, m^*)$ OUTPUT 1	
OUTPUT 0	

Fig. 4: Procedures of the Security Game $G^{\mathsf{MM\text{-}eTCR}}_{\mathcal{A},\mathsf{KH}}$.

Definition 3. *We say that a* KH *function family* H *is* $(T, \epsilon^{\mathsf{MM\text{-}eTCR}}_{\mathsf{KH}}, q)$*-secure if any polynomial-size adversary* $\mathcal{A}$ *runs in time* $T = T(\kappa)$ *and makes at most* $q = q(\kappa)$ *queries to* Proc.HChallenge$(\cdot)$*, and* $\Pr[G^{\mathsf{MM\text{-}eTCR}}_{\mathcal{A},\mathsf{KH}}(\kappa, q) = 1] \leq \epsilon^{\mathsf{MM\text{-}eTCR}}_{\mathsf{KH}}(\kappa)$.

Proc.Init() :	**Proc.Finalize(j, m^*) :**
$pms_{\mathsf{KH}} \leftarrow \mathsf{H.Setup}(1^\kappa)$,OUTPUT pms_{KH}	IF $\exists m_j \in$ HL *s.t.* $m_j \neq m^*$ $\bigwedge \mathsf{H}(hk, m_j) = \mathsf{H}(hk, m^*)$ OUTPUT 1
Proc.HChallenge(hk) :	OUTPUT 0
FOR $i \in [q] : m_i \xleftarrow{\$} \mathcal{M}_{\mathsf{KH}}$, APPEND $m_i \rightarrow$ HL, OUTPUT HL	

Fig. 5: Procedures of the Security Game $G^{\mathsf{SM\text{-}SPR}}_{\mathcal{A},\mathsf{KH}}$.

Definition 4. *We say that a* KH *function family* H *is* $(T, \epsilon^{\mathsf{SM\text{-}SPR}}_{\mathsf{KH}}, q)$*-secure if any polynomial-size adversary* $\mathcal{A}$ *runs in time* $T = T(\kappa)$ *and asks at most* $q = q(k)$ *queries to* Proc.HChallenge$(\cdot)$*, and* $\Pr[G^{\mathsf{SM\text{-}SPR}}_{\mathcal{A},\mathsf{KH}}(\kappa, q) = 1] \leq \epsilon^{\mathsf{SM\text{-}SPR}}_{\mathsf{KH}}(\kappa)$.

Proc.Init(hk) :	**Proc.Finalize(m^*) :**
$pms_{\mathsf{KH}} \leftarrow \mathsf{H.Setup}(1^\kappa)$, $m \xleftarrow{\$} \mathcal{M}_{\mathsf{KH}}$	IF $\mathsf{H}(hk, m) = \mathsf{H}(hk, m^*)$, OUTPUT 1
OUTPUT $pms_{\mathsf{KH}}, \mathsf{H}(hk, m)$	OUTPUT 0

Fig. 6: Procedures of the Security Game $G^{\mathsf{SS\text{-}OW}}_{\mathcal{A},\mathsf{KH}}$.

Definition 5. *We say that a* KH *function family* H *is* $(T, \epsilon^{\mathsf{SS\text{-}OW}}_{\mathsf{KH}})$*-secure if any polynomial-size adversary* $\mathcal{A}$ *runs in time* $T = T(\kappa)$ *and* $\Pr[G^{\mathsf{SS\text{-}OW}}_{\mathcal{A},\mathsf{KH}}(\kappa) = 1] \leq \epsilon^{\mathsf{SS\text{-}OW}}_{\mathsf{KH}}(\kappa)$.

Proc.Init() :	**Proc.Finalize(b^*) :**
$pms_{\mathsf{KH}} \leftarrow \mathsf{H.Setup}(1^\kappa)$, OUTPUT pms_{KH}	IF $b^* = b$, OUTPUT 1, ELSE OUTPUT 0
Proc.HChallenge(hk) :	
$b \xleftarrow{\$} \{0, 1\}$, $m \xleftarrow{\$} \mathcal{M}_{\mathsf{KH}}$, $r_0 \xleftarrow{\$} \mathcal{Y}_{\mathsf{KH}}$, $r_1 \leftarrow \mathsf{H}(hk, m)$, OUTPUT r_b	

Fig. 7: Procedures of the Security Game $G^{\mathsf{SS\text{-}UD}}_{\mathcal{A},\mathsf{KH}}$.

Definition 6. *We say that a* KH *function family* H *is* $(T, \epsilon^{\mathsf{SS\text{-}UD}}_{\mathsf{KH}})$*-secure if it holds that* $\left|\Pr[G^{\mathsf{SS\text{-}UD}}_{\mathcal{A},\mathsf{KH}}(\kappa) = 1] - \frac{1}{2}\right| \leq \epsilon^{\mathsf{SS\text{-}UD}}_{\mathsf{KH}}(\kappa)$ *for any polynomial-size adversary* $\mathcal{A}$ *running in polynomial time* $T = T(\kappa)$*.*

Tweakable Hash Functions. A tweakable hash (TH) function family $\mathsf{THF} = (\mathsf{Setup}, \mathsf{Eval})$ consists of two algorithms. $\mathsf{Setup}(1^\kappa)$ is a setup algorithm that takes as input the security parameter 1^κ and outputs a system parameter pms_{TH}. The evaluation algorithm $\mathsf{Eval}(P, \Gamma, m)$ takes as input a public parameter $P \in \mathcal{P}_{\mathsf{TH}}$, a tweak $\Gamma \in \mathcal{T}_{\mathsf{TH}}$, and a message $m \in \mathcal{M}_{\mathsf{TH}}$, where $\mathcal{P}_{\mathsf{TH}}$, $\mathcal{T}_{\mathsf{TH}}$, and $\mathcal{M}_{\mathsf{TH}}$ are the public parameter space, the tweak space and the message space, respectively. It outputs a hash value $\alpha \in \mathcal{Y}_{\mathsf{TH}}$, where $\mathcal{Y}_{\mathsf{TH}}$ is the hash value space.

We write $\mathsf{THF}(P,\Gamma,m)$ in place of $\mathsf{THF.Eval}(P,\Gamma,m)$ for simplicity. In a TH function, the public parameter might be deemed as a function key or index, and the tweak might be considered as a nonce [32]. Previous work [32] has discussed several security properties of a THF in the context of distinct tweaks. We assume that a THF satisfies two existing properties required by our PoPA construction, i.e., *single-function single-target message one-wayness* (SS-MOW), and *single-function single-target undetectability* (SS-UD).

In the sequel, we review two existing properties required by our PoPA construction, i.e., *single-function single-target message one-wayness* (SS-MOW), and *single-function single-target undetectability* (SS-UD).

Proc.Init() :	**Proc.Finalize(b^*)** :
$pms_{\mathsf{TH}} \leftarrow \mathsf{THF.Setup}(1^\kappa)$, $P \xleftarrow{\$} \mathcal{P}_{\mathsf{TH}}$ OUTPUT pms_{TH}, P	OUTPUT $b^* = b$
Proc.THChallenge(Γ) :	
$b \xleftarrow{\$} \{0,1\}$, $m \xleftarrow{\$} \mathcal{M}_{\mathsf{TH}}$, $r_0 \xleftarrow{\$} \mathcal{Y}_{\mathsf{TH}}$, $r_1 \leftarrow \mathsf{THF}(P,\Gamma,m)$, OUTPUT r_b	

Fig. 8: Procedures of the Security Game $G^{\mathsf{SS\text{-}UD}}_{\mathcal{A},\mathsf{THF}}$.

Definition 7. *We say that a* TH *function family* THF *is* $(T,\epsilon^{\mathsf{SS\text{-}UD}}_{\mathsf{TH}})$*-secure if* $\left|\Pr[G^{\mathsf{SS\text{-}UD}}_{\mathcal{A},\mathsf{THF}}(\kappa)=1]-\frac{1}{2}\right| \le \epsilon^{\mathsf{SS\text{-}UD}}_{\mathsf{TH}}$ *for any polynomial-size adversary* $\mathcal{A}$ *running in time* T *and making only one classical query to* Proc.THChallenge.

Proc.Init(Γ) :	**Proc.Finalize(m^*)** :
$pms_{\mathsf{TH}} \leftarrow \mathsf{THF.Setup}(1^\kappa)$ $P \xleftarrow{\$} \mathcal{P}_{\mathsf{TH}}$, $m \xleftarrow{\$} \mathcal{M}_{\mathsf{TH}}$ OUTPUT $pms_{\mathsf{TH}}, P, \mathsf{THF}(P,\Gamma,m)$	IF $\mathsf{THF}(P,\Gamma,m) = \mathsf{THF}(P,\Gamma,m^*)$ OUTPUT 1 OUTPUT 0

Fig. 9: Procedures of the Security Game $G^{\mathsf{SS\text{-}MOW}}_{\mathcal{A},\mathsf{THF}}$.

Definition 8. *We say that a* TH *function family* THF *is* $(T,\epsilon^{\mathsf{SS\text{-}MOW}}_{\mathsf{TH}})$*-secure if for any polynomial-size adversary* $\mathcal{A}$ *running in time* T*, it holds that* $\Pr[G^{\mathsf{SS\text{-}MOW}}_{\mathcal{A},\mathsf{THF}}(\kappa)=1] \le \epsilon^{\mathsf{SS\text{-}MOW}}_{\mathsf{TH}}$.

Pseudorandom Generators. A pseudorandom generator $\mathsf{PRG}=(\mathsf{Setup},\mathsf{Gen})$ consists of two algorithms. $\mathsf{Setup}(1^\kappa)$ takes input 1^κ and outputs pms_{PRG}, defining the seed space $\mathcal{S}_{\mathsf{PRG}}$ and range space $\mathcal{R}_{\mathsf{PRG}}$. $\mathsf{Gen}(s)$ generates a pseudorandom value $r \in \mathcal{R}_{\mathsf{PRG}}$ from a random seed $s \xleftarrow{\$} \mathcal{S}_{\mathsf{PRG}}$. The security of PRG requires that no efficient algorithm can distinguish $r=\mathsf{Gen}(s)$ from a random value.

We define a security game $G^{\mathsf{IND}}_{\mathcal{A},\mathsf{G}}(\kappa)$ that is played between an adversary $\mathcal{A}$ and a challenger based on a pseudorandom generator G and the parameter κ. The procedures of $G^{\mathsf{IND}}_{\mathcal{A},\mathsf{G}}(\kappa)$ are defined in Figure 10, in which the procedure Proc.Challenge can be asked at most once.

Proc.Init() :	**Proc.Finalize(b^*)** :
OUTPUT $\mathsf{G.Setup}(1^\kappa)$	IF $b^* = b$, OUTPUT 1, ELSE OUTPUT 0
Proc.Challenge() :	
$s \xleftarrow{\$} \mathcal{S}_{\mathsf{PRG}}$, $b \xleftarrow{\$} \{0,1\}$, $r_0 \xleftarrow{\$} \mathcal{R}_{\mathsf{PRG}}$, $r_1 \leftarrow \mathsf{G.Gen}(s)$, OUTPUT r_b	

Fig. 10: Procedures of the Security Game $G^{\mathsf{IND}}_{\mathcal{A},\mathsf{G}}$.

Definition 9. *We say that* G *is a* $(T,\epsilon^{\mathsf{IND}}_{\mathsf{PRG}})$*-secure pseudorandom generator if for any polynomial-size adversary* $\mathcal{A}$ *running in time* T*, it holds that* $\left|\Pr[G^{\mathsf{IND}}_{\mathcal{A},\mathsf{G}}(\kappa)=1]-\frac{1}{2}\right| \le \epsilon^{\mathsf{IND}}_{\mathsf{PRG}}$.

Merkle Tree. A Merkle tree scheme $\mathsf{MT}=(\mathsf{Setup},\mathsf{Build},\mathsf{GetPrf},\mathsf{Verify})$ includes four algorithms. $\mathsf{Setup}(1^\kappa)$ outputs parameters pms_{MT}, defining leaf space $\mathcal{L}_{\mathsf{M}}$ and proof space $\mathcal{PF}_{\mathsf{M}}$. $\mathsf{Build}(\{\mathrm{lf}_i\}_{i\in[N]})$ constructs a Merkle tree MI from N leaves in $\mathcal{L}_{\mathsf{M}}$ and outputs the initial state $\mathsf{st}_{\mathsf{BDS}}$ for the BDS algorithm [33]. $\mathsf{GetPrf}(\mathsf{st}_{\mathsf{BDS}},\mathrm{lf}_i)$ generates a proof $\mathrm{pf}_{\mathrm{lf}_i} \in \mathcal{PF}_{\mathsf{M}}$ for lf_i and updates $\mathsf{st}_{\mathsf{BDS}}$. $\mathsf{Verify}(\mathrm{MI.Rt},\mathrm{lf}_i,\mathrm{pf}_{\mathrm{lf}_i})$ validates lf_i's inclusion in MI.Rt, returning 1 if valid and 0 otherwise. Secure MT schemes ensure unforgeable proofs.

We define a security game $G^{\mathsf{UF}}_{\mathcal{A},\mathsf{M}}(\kappa)$ in Figure 11 for a Merkle tree scheme M. A secure Merkle tree scheme should prevent any adversary from forging the Merkle proof for a leaf node that does not belong to the Merkle tree.

Proc.Init($\{\mathrm{lf}_i\}_{i\in[N]}$) :	**Proc.Finalize($\mathrm{lf}^*,\mathrm{pf}^*$)** :
$pms_{\mathsf{M}} \leftarrow \mathsf{M.Setup}(1^\kappa)$ $\mathrm{MI} \leftarrow \mathsf{M.Build}(\{\mathrm{lf}_i\}_{i\in[N]})$ OUTPUT pms_{M}, MI	IF $1 \leftarrow \mathsf{M.Verify}(\mathrm{MI.Rt},\mathrm{lf}^*,\mathrm{pf}^*) \bigwedge \mathrm{lf}^* \notin \{\mathrm{lf}_i\}_{i\in[N]}$ OUTPUT 1 OUTPUT 0

Fig. 11: Procedures of the Security Game $G^{\mathsf{UF}}_{\mathcal{A},\mathsf{M}}$.

Definition 10. *We say that* M *is a* $(T,\epsilon^{\mathsf{UF}}_{\mathsf{MT}})$*-secure* MT *if for any polynomial-size adversary* $\mathcal{A}$ *running in time* T*, it holds that* $\Pr[G^{\mathsf{UF}}_{\mathcal{A},\mathsf{M}}(\kappa)=1] \le \epsilon^{\mathsf{UF}}_{\mathsf{MT}}$.

WOTS+C. It is a variant of the Winternitz One-Time Signature (WOTS) framework [16]. The construction idea behind WOTS is based on iteratively applying a one-way function to generate chaining signature values. WOTS+C uses a Winternitz parameter w to indicate the length of each signature chain. The number of signature chains in WOTS+C is $\ell = \left\lceil \frac{\kappa}{\log(w)} \right\rceil$. Instead of signing a message m, the WOTS+C scheme signs the hash value $\mathsf{H}_4(m\|sa)$, where sa is a short random salt. Two additional parameters $S_{w,\kappa}$ and $\ell_0 \in \mathbb{N}$ are introduced to let a base-w representation $(a_1,\ldots,a_\ell)$ of the value $\mathsf{H}_4(m\|sa)$ satisfy the *compact properties*: $\sum_{i=1}^{\ell} a_i = S_{w,\kappa}$ and $\forall i \in [\ell_0]$, $a_i = 0$. Let $C(n,v) = n!/v!(n-v)!$ be the combination function. The salt-finding probability [34] is denoted by $p_v = (\sum_{j=0}^{\ell}(-1)^j \cdot C(\ell,j)C(S_{w,\kappa}+\ell-jw-1,\ell-1)/(w^\ell 2^{\ell_0})$.

The WOTS+C scheme only signs and verifies $\ell_1 = \ell - \ell_0$ signature-chains, each of which makes $w-1$ calls to a chaining function. Here, we consider a tweakable hash function THF as the chaining function to construct the chain structures. Let $\Gamma_{i,j}$ denote the tweak associated with the j-th function call in the i-th signature-chain. Meanwhile, the tweaks for the $(w-1)\ell_1$ function calls must be distinct to guarantee the security of THF. For $i \in [\ell_1]$, SKE^0_i is the initial signing-key element in the i-th signature-chain and the $(j+1)$-th one is computed as $\mathsf{SKE}^j_i := \mathsf{THF}(P,\Gamma_{i,j},\mathsf{SKE}^{j-1}_i)$ when $j \in [w-1]$.

IV. Security Model of PoPA

Basic Notions. PoPA operates as a three-party protocol involving a client device (iC), a verifier (iS), and an auditor (iA). The client device acts as an aliveness prover, continuously

generating and sending aliveness proofs to the verifier, which checks these proofs and reports any death events. For a persistently alive client, the aliveness proofs must confirm its running status at any historical time. Meanwhile, an honest auditor periodically checks the verifier's historical verification states to ensure the verifier has performed its duties faithfully. We define the syntax of PoPA via four algorithms $\Sigma = (\mathsf{Setup}, \mathsf{ProofGen}, \mathsf{Verify}, \mathsf{Audit})$ as:

• $(sk_{\mathsf{iC}}, \mathsf{Pms}, \pi_{\mathsf{iC}}) \leftarrow \mathsf{Setup}(1^\kappa, \mathsf{SetPms})$: The setup algorithm takes as input the security parameter 1^κ and setup parameters $\mathsf{SetPms} = (\tilde{T}_{ps}, \tilde{T}_a, \tilde{T}_{ls}, \mathsf{aux})$, where $\tilde{T}_{ps}$ is the start time of a protocol instance, $\tilde{T}_a$ denotes the upper bound of the time length covered by an aliveness proof (which should be specified by the application using PoPA, e.g., 3 minutes [2]), $\tilde{T}_{ls}$ denotes the life span of a PoPA instance, and aux denotes other auxiliary parameters (that may be needed by a specific scheme). This algorithm generates the initial secret key $sk_{\mathsf{iC}} \in \mathcal{SK}_{\mathsf{PoPA}}$ of the client iC, the system parameter Pms and the initial verification point π_{iC}. Meanwhile, iC securely stores the initial secret key sk_{iC}. The client and the verifier would also keep local mutable state variables $\mathsf{st}_{\mathsf{iC}}$ and $\mathsf{st}_{\mathsf{iS}}$, respectively. We let $T_{be} \in \mathsf{st}_{\mathsf{iC}}$ be the end time of the last proof being generated by iC, and $T_{ack} \in \mathsf{st}_{\mathsf{iS}}$ be the check time of the last proof verified by the verifier. In addition, we let $\Delta_f \in \mathsf{Pms}$ be a small constant *failure-tolerant time* (e.g., for modeling the time of occasional rebooting, task switching, I/O jamming, and tolerable network delay) such that $\Delta_f << \tilde{T}_a$. It can be measured by any network/system experts [35], that is out-of-the-scope of this paper. The other contents of Pms, $\mathsf{st}_{\mathsf{iC}}$, and $\mathsf{st}_{\mathsf{iS}}$ are supposed to be defined by concrete schemes.

• $\rho \leftarrow \mathsf{ProofGen}(sk_{\mathsf{iC}}, \mathsf{st}_{\mathsf{iC}}, T_c)$: The proof generation algorithm takes as input the secret key sk_{iC}, the local state $\mathsf{st}_{\mathsf{iC}}$ of iC, and the current system time T_c. It outputs the latest aliveness proof ρ if $T_{be} \leq T_c < T_{be} + \Delta_f$ and the time spent on generating the new aliveness proof is smaller than the time-length $\tilde{T}_a$. Otherwise, the algorithm outputs a failure $\perp$ (implying the break of aliveness). If a proof is successfully generated, this algorithm may update iC's secret key sk_{iC} and other values in state $\mathsf{st}_{\mathsf{iC}}$.

• $\{0,1\} \leftarrow \mathsf{Verify}(\pi_{\mathsf{iC}}, \mathsf{st}_{\mathsf{iS}}, \rho, T_r)$: The proof verification algorithm takes as input the initial verification point π_{iC} and local state $\mathsf{st}_{\mathsf{iS}}$ to verify the aliveness proof ρ received at time T_r, and outputs 1 iff ρ is valid, and 0 otherwise. Meanwhile, $\mathsf{st}_{\mathsf{iS}}$ may be updated by this algorithm, e.g., appending ρ to the state $\mathsf{st}_{\mathsf{iS}}$.

• $\{0,1\} \leftarrow \mathsf{Audit}(\pi_{\mathsf{iC}}, \mathsf{st}_{\mathsf{iS}}, \mathsf{st}_{\mathsf{iA}}, T)$: The audit algorithm takes as input the initial verification point π_{iC}, the local state $\mathsf{st}_{\mathsf{iS}}$ of iS, and a state $\mathsf{st}_{\mathsf{iA}}$, and outputs 1 iff $\mathsf{st}_{\mathsf{iS}}$ is a valid state for verifying the aliveness proof at time T, and 0 otherwise.

Correctness. We say that a PoPA scheme is correct if, given a time T (s.t., $T_{be} \leq T < T_{be} + \Delta_f$) and the state $\mathsf{st}_{\mathsf{iC}}^T$ (s.t., $T_{be} \in \mathsf{st}_{\mathsf{iC}}^T$) of the client at T, the client can compute $\rho := \mathsf{ProofGen}(sk_{\mathsf{iC}}, \mathsf{st}_{\mathsf{iC}}^T, T)$ within time $\tilde{T}_a$, and it holds that $\mathsf{Verify}(\pi_{\mathsf{iC}}, \mathsf{st}_{\mathsf{iS}}^{T_r}, \rho, T_r) = 1$ and $\mathsf{Audit}(\pi_{\mathsf{iC}}, \mathsf{st}_{\mathsf{iS}}^{T_r}, \mathsf{st}_{\mathsf{iA}}^{T_r}, T_r) = 1$ with a receiving time T_r (s.t., $T_{ack} < T_r < T_{ack} + \tilde{T}_a + \Delta_f$) and the state $\mathsf{st}_{\mathsf{iS}}^{T_r}$ of iS at T_r. As long as proofs are promptly generated within the time limit by the ProofGen algorithm and received correctly in time, they should pass verification. However, any network errors or attacks (e.g., dropping proofs) that remain unresolved beyond a reasonable time Δ_f must be treated as a breach of aliveness.

Security Definition. Following the game-based approach in [36], [37], we define two security games for PoPA to formalize the new security properties of *persistent aliveness* (PAlive) and *audit* (Audit). Let $G^{\mathsf{PoPA}}_{\mathcal{A},\Sigma}(\kappa, \mathsf{SetPms}, \mathsf{exp})$ denote the game $\mathsf{exp} \in \{\mathsf{PAlive}, \mathsf{Audit}\}$ that an adversary $\mathcal{A}$ plays against the PoPA scheme Σ with parameters κ and $\mathsf{SetPms} = (\tilde{T}_{ps}, \tilde{T}_a, \tilde{T}_{ls}, \mathsf{aux})$. These games share a series of procedures defined in Figure 12, which are simulated by a challenger. The adversary $\mathcal{A}$ starts a game by calling Proc.Init and ends the game with Proc.Finalize. All other procedures can be queried sequentially and adaptively by the adversary. $\mathcal{A}$ can call Proc.GetNextPf to get an aliveness proof (to model known proofs attacks) of the client iC at the current time T_c, and send a proof ρ on behalf of iC through a Proc.ReceivePf query for verification (e.g., testing its forgeries). The Proc.Execute procedure is used to model any delivery failures (e.g., caused by network attacks or equipment troubles) of honestly generated aliveness proofs. We model the corruption of the client by the Proc.Corrupt procedure and the compromise of any party $\mathsf{ID} \in \{\mathsf{iC}, \mathsf{iS}, \mathsf{iA}\}$ to learn its state $\mathsf{st}_{\mathsf{ID}}$ via the Proc.ReState procedure. Meanwhile, we adopt the standard pre-compromise model. Namely, once the adversary learns the prover's secret key or protected state, subsequent impersonation is trivial and out of scope. In our security game (Figure 12), pre-compromise is captured by the secret-key disclosure time T_{sk} and the state disclosure time T_{st}. We say that an adversary's compromise behaviors violate pre-compromise rule if disclosure occurs before the time challenge begin at time sT (i.e., with constraint $(T_{sk} < \mathsf{sT}) \vee (T_{st} < \mathsf{sT})$) or before the targeted window ends (i.e., with constraint $(T_{sk} < T^* + \tilde{T}_a + \Delta_f) \vee (T_{st} < T^* + \tilde{T}_a + \Delta_f)$), where T^* is the adversary's committed target time for a forgery.

A stronger compromise scenario is an attacker who reprograms a device to continue producing accepting proofs while disabling core functionality. We treat this as post-compromise and therefore outside our security formalism. Functional correctness is orthogonal and should be enforced by complementary mechanisms such as secure boot [38] and remote attestation [39]. Accordingly, PoPA is formulated under the assumption that, within each aliveness-proof window, the device adheres to the intended code path.

To model time, we define the procedure Proc.GetCT to retrieve the current time (stored in the variable current_time). Proc.GetCT is implemented with an additional tape on the Turing Machine, dedicated to counting the elementary operations used by the challenger. We assume that the values of the variable current_time are monotonically increasing. But it does not count the running time of the adversary that it executes locally (i.e., those steps without explicitly included in the challenger's tap.

We say the adversary $\mathcal{A}$ wins a game if and only if the output of Proc.Finalize is 1.

Proc.Init($\mathsf{TC}_{\mathcal{A}}$) :
$(sk_{\mathsf{iC}}, \mathsf{Pms}, \pi_{\mathsf{iC}}) \leftarrow \Sigma.\mathsf{Setup}(1^{\kappa}, \mathsf{SetPms})$
$\mathsf{isTCh} := 0$, $\mathsf{isTWin} := 0$
$\mathsf{PFL} := \emptyset$, $T_{sk} := T_{st} := \infty$
OUTPUT $\mathsf{Pms}, \pi_{\mathsf{iC}}$

Proc.Execute() :
$T_c := \mathsf{Proc.GetCT}()$
$\rho := \Sigma.\mathsf{ProofGen}(sk_{\mathsf{iC}}, \mathsf{st}_{\mathsf{iC}}, T_c)$
APPEND $(\rho, T_c) \rightarrow \mathsf{PFL}$
IF $\rho \neq \perp$ OUTPUT 1, ELSE OUTPUT 0

Proc.GetNextPf() :
$T_c := \mathsf{Proc.GetCT}()$
$\rho := \Sigma.\mathsf{ProofGen}(sk_{\mathsf{iC}}, \mathsf{st}_{\mathsf{iC}}, T_c)$
APPEND $(\rho, T_c) \rightarrow \mathsf{PFL}$
OUTPUT ρ

Proc.ReState(ID) :
IF ID = iC and $T_{st} = \infty$, $T_{st} := \mathsf{Proc.GetCT}()$
IF ID $\in \{\mathsf{iC}, \mathsf{iS}, \mathsf{iA}\}$, OUTPUT $\mathsf{st}_{\mathsf{ID}}$
OUTPUT $\perp$

Proc.Corrupt() :
IF $T_{sk} = \infty$, $T_{sk} := \mathsf{Proc.GetCT}()$, OUTPUT sk_{iC}

Proc.GetCT() :
OUTPUT current_time

Proc.ReceivePf(ρ) :
$T_r := \mathsf{Proc.GetCT}()$
OUTPUT $\Sigma.\mathsf{Verify}(\pi_{\mathsf{iC}}, \mathsf{st}_{\mathsf{iS}}, \rho, T_r)$

Proc.TChallenge() :
IF isTCh = 1, OUTPUT $\perp$ ▷ This query can be asked only once
isTCh := 1, sT := Proc.GetCT(), $\rho^* \leftarrow \mathsf{TC}_{\mathcal{A}}(sk_{\mathsf{iC}}, \mathsf{st}_{\mathsf{iC}}, \mathsf{sT})$, eT := Proc.GetCT() ▷ It gets the runtime of $\mathsf{TC}_{\mathcal{A}}$
IF $(T_{sk} < \mathsf{sT}) \vee (T_{st} < \mathsf{sT}) \vee (\exists T' \in \mathsf{PFL}$ s.t. $T' \geq \mathsf{sT} - \tilde{T}_a)$ ▷ T_{sk}: corrupt time; T_{st}: state reveal time
OUTPUT $\perp$ ▷ It excludes trivial attacks on getting sk, state, and target proof before challenge
IF $\mathsf{eT} - \mathsf{sT} < \tilde{T}_a^{\tau} \wedge \Sigma.\mathsf{Verify}(\pi_{\mathsf{iC}}, \mathsf{st}_{\mathsf{iS}}, \rho^*, \mathsf{eT}) = 1$, isTWin := 1 ▷ $\mathcal{A}$ wins if $\mathsf{TC}_{\mathcal{A}}$ is indeed faster
OUTPUT isTWin ▷ The time constraint is derived from the properties of PoW [20], [21].

Proc.Finalize($\mathsf{st}^*_{\mathsf{iS}}, \rho^*, T^*$) :
$T_c := \mathsf{Proc.GetCT}()$
IF exp = PAlive $\wedge$ isTWin = 1, OUTPUT 1 ▷ $\mathcal{A}$ wins the time challenge
IF isTCh = 1, OUTPUT $\perp$ ▷ $\mathcal{A}$ fails time challenge
IF $(\rho^* \in \mathsf{PFL}) \vee (T_{sk} < T^* + \tilde{T}_a + \Delta_f) \vee (T_{st} < T^* + \tilde{T}_a + \Delta_f)$
OUTPUT $\perp$ ▷ It excludes trivial attacks on getting sk, state, and target proof
IF $(\mathsf{exp} = \mathsf{PAlive} \wedge \Sigma.\mathsf{Verify}(\pi_{\mathsf{iC}}, \mathsf{st}_{\mathsf{iS}}, \rho^*, T_c) = 1) \vee (\mathsf{exp} = \mathsf{Audit} \wedge \Sigma.\mathsf{Audit}(\pi_{\mathsf{iC}}, \mathsf{st}^*_{\mathsf{iS}}, \mathsf{st}_{\mathsf{iA}}, T^*) = 1$
$\wedge\ \rho^* \in \mathsf{st}^*_{\mathsf{iS}})$, OUTPUT 1 ▷ $\mathcal{A}$ successfully forges either a proof or a state
OUTPUT 0

Fig. 12: Procedures of the Security Game $G^{\mathsf{PoPA}}_{\mathcal{A},\Sigma}$ for a PoPA Scheme Σ.

Meanwhile, $\mathcal{A}$ can break PAlive by either asking a Proc.TChallenge query to produce an aliveness proof significantly faster than running ProofGen algorithm or submitting a valid forgery (ρ^*, T^*) of an uncompromised client in the Proc.Finalize query. The Proc.TChallenge procedure tests the robustness of execution time indicated by the aliveness proof. It does so by running a polynomial-size malicious circuit $\mathsf{TC}_{\mathcal{A}}$, representing the Σ.ProofGen algorithm, which is chosen and specified by the adversary during the Proc.Init query. Note that, via the invoking time of the Proc.TChallenge procedure, the adversary can indirectly specify the secret key and secret states used to execute $\mathsf{TC}_{\mathcal{A}}$. Furthermore, we restrict $\mathcal{A}$ so that after querying Proc.TChallenge (permitted only once by checking the isTCh variable), it can only invoke Proc.Finalize. The Audit property requires that $\mathcal{A}$ must not be able to generate malicious verification state $\mathsf{st}^*_{\mathsf{iS}}$ encompassing an aliveness proof ρ^* capable of passing verification but not originating from any Proc.GetNextPf queries, even if corresponding the client is compromised after ρ^* was generated.

Definition 11. *We say that a polynomial-size adversary $\mathcal{A}$ $(T_{\mathcal{A}}, \epsilon^{\mathsf{exp}}_{\mathsf{PoPA}})$-breaks the PoPA scheme Σ with parameters $(\tau, \kappa, \mathsf{SetPms})$ if $\mathcal{A}$ runs in time $T_{\mathcal{A}}$ and $\Pr[G^{\mathsf{PoPA}}_{\mathcal{A},\Sigma}(\kappa, \mathsf{SetPms}, \mathsf{exp}) = 1] \geq \epsilon^{\mathsf{exp}}_{\mathsf{PoPA}}$, where $\mathsf{exp} \in \{\mathsf{PAlive}, \mathsf{Audit}\}$ and $\tau < 1$. We say that Σ is $(T_{\mathcal{A}}, \epsilon^{\mathsf{exp}}_{\mathsf{PoPA}})$-secure if no adversaries $(T_{\mathcal{A}}, \epsilon^{\mathsf{exp}}_{\mathsf{PoPA}})$-breaks it.*

In Table I, we summarize the differences between the PoPA model and the PoA model [2] from the following perspectives. Our PoPA model is strengthened from the previous PoA model by modeling more security properties and adversarial capabilities. An innovative aspect of our PoPA model lies in the formulation of the new Audit property compared to the PoA model. Namely, the PoA model fails to cover the threats caused by malicious verifiers who fail to accomplish their aliveness-verifying and alerting duties. Additionally, the definition of the aliveness property diverges between the two models. The PoA model confines itself to intermittent aliveness, grounded in discrete time slots with fixed intervals. In contrast, our PoPA model embraces continuous time, facilitating the modeling of continuous aliveness proofs. In particular, we allow an adversary to challenge the generation time of an aliveness proof via the Proc.TChallenge procedure, which has not been considered before. Moreover, our model considers the leakage of states (which may contain credentials) of the entities. In addition, the Proc.Corrupt procedure is prohibited from asking against the challenged client in the PoA model, so it cannot model forward secrecy (FS) as ours. In essence, these new procedures enable the PoPA model to be stronger than the PoA model.

TABLE I: Model Comparison

Model	Security Properties		Adversarial Capabilities				
			Differences				Similarities
	Aliveness	Audit	RState	Execute	TChallenge	Corrupt	
PoA	Intermittent	×	×	×	×	Non-FS	Init, GetNextPf ReceivePf
PoPA	Continuous	√	√	√	√	FS	

V. A Scalable PoPA scheme SPAC

This section defines SPAC, which constitutes a scalable proof of aliveness chain.

A. Construction Problems and Techniques

We introduce a toy PoPA construction to illustrate key design challenges. This scheme leverages a HPoW mechanism to generate sequential aliveness proofs, replacing trusted time-stamping with computational effort. During execution, the verifier relies solely on a locally reliable clock to record the acknowledgment time T_{ack} of the most recent valid HPoW solution. By ensuring that consecutive valid proofs arrive within the time window $(T_{ack}, T_{ack} + \tilde{T}_a + \Delta_f)$, where $\tilde{T}_a$ represents the maximum time allowed to solve an HPoW puzzle, the verifier can confirm persistent aliveness without needing the client's actual clock time. As a result, time synchronization is only needed during the initial setup.

To authenticate puzzles, it employs a Merkle tree, where leaf preimages serve as credentials for HPoW puzzle generation. As a result, an attacker without knowledge of these credentials cannot solve the puzzle before the client completes it and releases the corresponding proof, which contains the credential. In other words, even with greater computational power, an attacker cannot preemptively solve the HPoW puzzle ahead of the client. Additionally, a pseudorandom generator evolves one-time credentials over time, ensuring forward secrecy and enabling Audit.

Technical Challenges. To achieve a lightweight PoPA, we aim to remove unnecessary security features, such as message authentication. Thus, in the above toy example, we only authenticate the generation of the HPoW puzzle, not the HPoW solution, enabling the verifier to check only that the puzzle was generated by the corresponding client. However, this introduces a challenge: can we construct a lightweight PoPA scheme, following this approach, without relying on message authentication? This constraint inherently excludes the direct use of signature schemes for building PoPA.

We further explore the properties of HPoW to tackle this challenge. An important observation is the security concern when using arbitrary HPoW schemes, particularly randomized ones, due to the substantial computational power gap between resource-constrained clients and powerful attackers. To accommodate the client's limited computational capacity, we must set a relatively low difficulty level for HP, which inadvertently creates an easily solvable puzzle for the adversary. This gives rise to what we refer to as *solution manipulation attacks*, where an adversary, upon observing a legitimate aliveness proof, could potentially forge a valid aliveness proof (either the current proof or historical proof in the verifier's state) by computing an alternative valid HPoW solution for the same puzzle. The root cause lies in the non-determinism of randomized HPoW schemes, which allow multiple valid solutions for a single puzzle. A potential solution is to employ a deterministic HPoW scheme, for thwarting such solution manipulation attacks. In particular, if the solution inherently represents the number of hash operations performed, it enables the estimation of the client's execution time without relying on timestamps, thereby mitigating the difficulty of correlating hash-based work with precise timestamps. In principle, any difficulty function that faithfully measures the prover's objective running time can be adopted (with minimal impact on primary system functions), and this choice is independent of the attacker's computational power. Consequently, PoPA can be executed opportunistically during idle periods and does not materially interfere with primary system functions.

Furthermore, a shortcoming of the above toy construction is its lack of scalability. The need for more credentials to extend the lifespan leads to increased computation and proof size due to a larger Merkle tree. This raises the critical question of replenishing aliveness proofs without imposing a heavier burden on the client. However, *realizing aliveness proof replenishment with the* Audit *property remains an open challenge* (as discussed in Section I).

In this context, two key challenges are addressed. One major technical challenge (**TC**1) involves circumventing the Audit issue in Jin et al.'s OTS-based replenishment scheme, preventing malicious verifiers from exploiting signature values to manipulate aliveness proofs. The other one (**TC**2) focuses on maintaining stable overheads, such as proof generation time and size, avoiding spikes. This requires a new OTS integration, unlike Jin et al.'s scheme [2], which embeds large 256×256-bit OTS signatures in a single proof.

Further Construction Ideas. SPAC integrates a tailored OTS scheme with the toy construction to securely replenish credentials. To overcome challenges (**TC**1 and **TC**2), SPAC employs a variant of the WOTS+C scheme beneath a Merkle tree, forming its core authenticated credential structure (ACS). We show the high-level overview of the ACS in Figure 13. Each signature-chain in the WOTS+C variant links to a Merkle tree leaf, enabling a cohesive structure where the client can sign the Merkle root of a new ACS instance for replenishment. Unlike naively using an FSDS signature over an HPoW solution, which wastes many signature elements on a single HPoW puzzle, SPAC optimizes credential usage by maximizing the values used on the signature-chain.

We observe that the original WOTS+C signature-chain is vulnerable to Audit attacks because other signing-key elements can be computed from corresponding signature values, making them unsuitable as authentication credentials. Each missing credential must offer forward secrecy, preventing recovery by the verifier using its state. To achieve this and address **TC**1, we modify the WOTS+C signature-chain with private tweaks instead of public ones, resulting in a variant WOTS+CP. We use a PRG to chain these private tweaks, computing them in reverse order from the signature-chain's growth (cf. Figure 13). This ensures that missing tweaks remain unrecoverable from the proofs. These private tweaks serve as credentials for HPoW computation. The client starts with the preimage of a Merkle tree leaf, followed by signing-key elements in reverse order until reaching the signature value. The verifier checks the final signature value, then accepts the Merkle root of the new ACS instance. This approach distributes the computation of the new ACS instance across aliveness proof generation, minimizing overhead and balancing the computational load. The replenishment feature keeps the top Merkle tree in each ACS instance compact, significantly reducing proof size (compared to the toy example and naive FSDS-based schemes).

B. Construction Overview

Initially, a TTP runs the Setup algorithm to establish global parameters—start time, proof interval, lifespan, and auxiliary values for credential generation and proof verification—and to initialize all cryptographic building blocks. It also constructs two bootstrap ACS instances (as illustrated in Figure 13), from which the client will derive and later replenish one-time credentials. The client, verifier, and auditor each initialize their local state, tracking indices of generated and verified proofs, credential-chain positions, and acknowledgment times.

When the ProofGen algorithm is invoked, the client draws a precomputed credential and a signing-key element from its current ACS instance to form a challenge message and

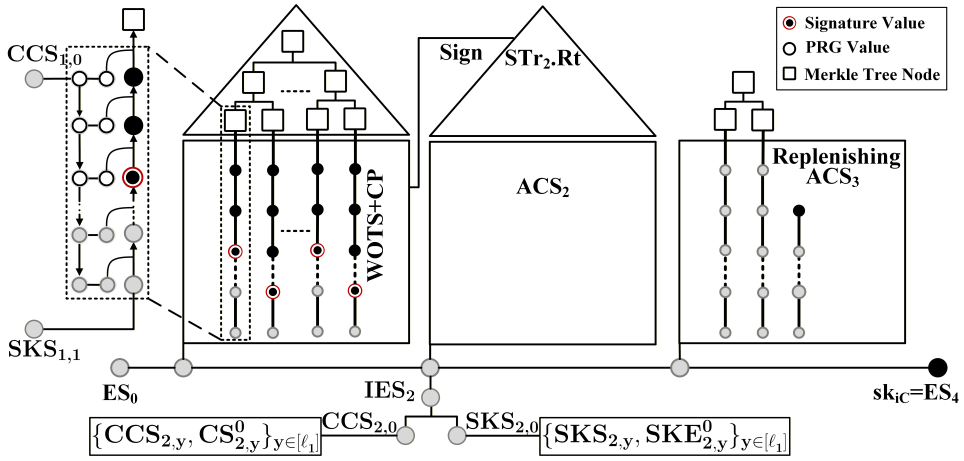


Fig. 13: Overview of SPAC's Authenticated Credential Structure (ACS). Hollow circles are PRG-derived credential chain (seeds and credentials). Black solid points are used signing-key elements (SKEs). SKEs with red edges are signature values, determined by the Merkle root of the next ACS instance.

then generate a deterministic HPoW puzzle. It assembles the aliveness proof by including the solution to this puzzle, attaching a Merkle proof or an ACS salt when crossing a chain or epoch boundary. Meanwhile, one of three "switch tasks" is performed for each proof: i) advancing within the same chain while unused credentials remain; ii) switching to the next chain; or iii) rolling over to the next epoch by transitioning to a new ACS instance.

Upon receipt, the Verify algorithm first checks that the proof's receiving time exceeds the previous acknowledgment and falls within the allowed PoW-based window (as briefly described above). It then, depending on the proof case, either hashes the credential and the signing-key element against the prior anchor, verifies a supplied Merkle proof, or validates the signature of the new ACS root by checking accumulated chain counts. If successful, the verifier advances the chain or epoch indices, updates the proof counter and acknowledgment time, and accepts the proof. Otherwise, it immediately rejects the proof due to verification failure or replay.

Finally, the Audit algorithm enables an external auditor to re-verify the stored proofs and timestamps. This ensures that each accepted proof was correctly validated and that no fraudulent or inconsistent state persists in the verifier's record.

C. Our New Building Blocks

In this section, we introduce the detailed construction of our two new building blocks that we will be used to realize the construction ideas of SPAC.

A Deterministic Hash-based Proof of Work. Our HPoW scheme relies on a TrCR function H. Notably, the truncation index i can be determined by the difficulty d of a puzzle, i.e., setting $i = d$. Our key observation (**KO1**) is to achieve determinism by constraining the identification of the "first" solution within a designated solution space, which both the prover and verifier can iteratively explore until a valid solution is found. While this approach raises the question of reducing the verifier's workload, we further observe (**KO2**) that if H is *regular* [40], with each range point having exactly two preimages, the solution space can be defined as twice the bit-size of the truncated hash range. This allows constructing a puzzle based on the first value in the solution space and a random challenge message, where solving involves finding a *unique* hash collision within this constrained space. Thus, if the solution lies within the reduced solution space, then the verifier does not need to re-solve the puzzle. For irregular hash functions, a wider solution space can accommodate potential solutions, ensuring validity by adopting **KO1** when **KO2** does not apply.

Specifically, the setup algorithm HPoW.Setup is run to generate a hash key $hk \xleftarrow{\$} \mathcal{K}_{\mathsf{H}}$ so that $pms_{\mathsf{HPoW}} := hk$. The difficulty space $\mathcal{D}_{\mathsf{PoW}}$ is defined by the bit-length of hash value space of H. We let $\mathsf{HPoW.Gen}(d, cm)$ generate a puzzle as $p := (d, cm, 0)$, given difficult d and a challenge-message cm. $\mathsf{HPoW.Solve}(p)$ relies on H to find an integer $s > 0$ such that $\mathsf{H}[d](cm||0) = \mathsf{H}[d](cm||s)$. It iterates over the solution space somehow (e.g., by incrementing $s := s+1$) until the above condition is satisfied. The verification algorithm $\mathsf{HPoW.Verify}(p, s')$ returns 1 if the above equality holds and one of the following conditions is met: i) if $s' \leq 2^{d+1}$; or ii) $s' > 2^{d+1}$ and no solution $s < s'$ exists (this requires the verifier to re-solve the puzzle itself). Our HPoW works well when d is not large. The ratio of step ii) is small (see footnote 6), so that the verifier can also leverage such a ratio to check any potential DoS attacks against the verifier.

Lemma 1. *Let* H *be a* $(q, d, T, \epsilon^d_{\mathsf{TrCR}})$*-secure* TrCRH *family and* $0 < \tau < 1$*, then* HPoW *is* $(d, T(d), \epsilon^d_{\mathsf{PoW}})$*-hard* PoW *scheme.*

Proof. It suffices to show that any adversary $\mathcal{A}$, which can solve a HPoW puzzle within time $T \leq T(d)^\tau$ and with a probability $\epsilon^d_{\mathsf{PoW}} > \epsilon^d_{\mathsf{TrCR}}$, can be used to construct an efficient algorithm $\mathcal{B}$ to break the TrCR of H. This reduction works since we define the procedure of finding a solution to be identical to find a d-bit truncated collision. $\mathcal{B}$ can simply run $\mathcal{A}$ as subroutine, and simulate the game for $\mathcal{A}$ as the HPoW challenger. Specifically, $\mathcal{B}$ can use the hash key hk obtained from the TrCR challenger to answer the Proc.Init

query from $\mathcal{A}$. For the Proc.PChallenge() query, it mainly samples a random challenge-message cm to generate the challenge-puzzle $p = (d, cm, 0)$. If $\mathcal{A}$ succeeds in submitting a solution s within the given time, then $\mathcal{B}$ can simply forward $(cm||0, cm||s)$ to the TrCR challenger to win the game. This can complete the proof. □

Private Tweakable Hash Functions. Previous THFs [32] utilize public tweaks, which are only required to be distinct. We need to adapt the security notion of THF to cover the security of tweaks, i.e., the tweak one-wayness in the context of *single-function single-target* (SS-TOW). Unlike traditional one-way functions, adversaries in the SS-TOW game of PTH can select the message to generate the challenge value.

A tweakable hash function that satisfies SS-TOW will be called as a private tweakable hash function (PTH). We formally define the new game $G^{\mathsf{SS\text{-}TOW}}_{\mathcal{A},\mathsf{THF}}(\kappa)$ in Figure 14.

Proc.Init(m) :	**Proc.Finalize(Γ^*)** :
$pms_{\mathsf{TH}} \leftarrow \mathsf{THF.Setup}(1^\kappa)$ $P \xleftarrow{\$} \mathcal{P}_{\mathsf{TH}}$, $\Gamma \xleftarrow{\$} \mathcal{T}_{\mathsf{TH}}$ OUTPUT $pms_{\mathsf{TH}}, P, \mathsf{THF}(P, \Gamma, m)$	IF $\mathsf{THF}(P, \Gamma, m) = \mathsf{THF}(P, \Gamma^*, m)$ OUTPUT 1 OUTPUT 0

Fig. 14: Procedures of the Security Game $G^{\mathsf{SS\text{-}TOW}}_{\mathcal{A},\mathsf{THF}}$.

Definition 12. *We say that a tweakable hash function family* THF *is* $(T, \epsilon^{\mathsf{SS\text{-}TOW}}_{\mathsf{PTH}})$*-secure if any polynomial-size adversary* $\mathcal{A}$ *runs in polynomial time* T *and* $\Pr[G^{\mathsf{SS\text{-}TOW}}_{\mathcal{A},\mathsf{THF}}(\kappa) = 1] \leq \epsilon^{\mathsf{SS\text{-}TOW}}_{\mathsf{PTH}}$.

For building SPAC, we also need PTH to satisfy *single-function multi-target extended target-collision resistance* (SM-eTCR), adapted from [34]. In the SM-eTCR game $G^{\mathsf{SM\text{-}eTCR}}_{\mathcal{A},\mathsf{THF}}(\kappa)$ (as defined in Figure 15), an adversary can manipulate the challenges by specifying the tweak used in challenges. Here, we restrict this control by requiring distinct tweaks as described in [32].

Proc.Init() :	**Proc.THChallenge(Γ, m)** :
$pms_{\mathsf{TH}} \leftarrow \mathsf{THF.Setup}(1^\kappa)$, $P \xleftarrow{\$} \mathcal{P}_{\mathsf{TH}}$, $cnt := 0$ OUTPUT pms_{TH}	$cnt := cnt + 1$ IF $cnt \leq q$ IF $cnt \neq q$, $\psi := \emptyset$ ELSE $\psi := P$ APPEND $(cnt, \Gamma, m) \rightarrow \mathsf{HL}$ OUTPUT $\mathsf{THF}(P, \Gamma, m), \psi$ OUTPUT $\perp$
Proc.Finalize(j, Γ^*, m^*) : IF $\exists(\Gamma_j, m_j) \in \mathsf{HL}$ *s.t.* $\mathsf{DIST}(\{\Gamma_u\}_{u\in[q]}) = 1$ $\bigwedge(\Gamma_j, m_j) \neq (\Gamma^*, m^*)$ $\bigwedge \mathsf{THF}(P, \Gamma_j, m_j) = \mathsf{THF}(P, \Gamma^*, m^*)$, OUTPUT 1 OUTPUT 0	

Fig. 15: Procedures of the Security Game $G^{\mathsf{SM\text{-}eTCR}}_{\mathcal{A},\mathsf{THF}}$.

Definition 13. *We denote the predicate* $\mathsf{DIST}(\{\Gamma_u\}_{u\in[q]}) = (\forall u, v \in [q], u \neq v) : \Gamma_u \neq \Gamma_v$*, i.e.,* $\mathsf{DIST}(\{\Gamma_u\}_{u\in[q]})$ *outputs* 1 *iff all tweaks are distinct. Then, we say that a* TH *function family* THF *is* $(T, \epsilon^{\mathsf{SM\text{-}eTCR}}_{\mathsf{PTH}}, q)$*-secure if any polynomial-size adversary* $\mathcal{A}$ *runs in time* T *and makes at most* q *classical queries to* Proc.THChallenge(·)*, and* $\Pr[G^{\mathsf{SM\text{-}eTCR}}_{\mathcal{A},\mathsf{THF}}(\kappa, q) = 1] \leq \epsilon^{\mathsf{SM\text{-}eTCR}}_{\mathsf{PTH}}$.

Here, we present two constructions of PTH that can provide SM-eTCR and SS-TOW and other security properties (i.e., SS-MOW and SS-UD) required by SPAC. Achieving all these properties simultaneously makes the construction of PTH quite challenging. The quantum generic security of SM-eTCR and SS-TOW is discussed in Appendix A (in supplemental materials).

PTH *Constructions.* The first construction is identical to the one proposed in [32], [34] using a plain and keyless hash function H_3. However, its weakness lies in the reliance on random oracle modeling of H_3 for the security analysis. In contrast, our second construction innovatively uses a KH function H_1 with specific standard security properties.

We assume that the public parameter space, the tweak space, and the message space of THF have bit lengths ℓ_P, ℓ_Γ, and $\ell_M = \ell_\Gamma$, respectively.

Construction 1 *([32], [34]):* Given a hash function $\mathsf{H}_3 : \{0,1\}^{\ell_P+2\ell_M} \rightarrow \{0,1\}^{\ell_M}$, we construct a PTH function THF as $\mathsf{THF}(P, \Gamma, M) = \mathsf{H}_3(P\|\Gamma\|M)$.

Construction 2*:* Given the two hash functions H_1: $\{0,1\}^{\ell_P} \times \{0,1\}^{2\ell_M} \rightarrow \{0,1\}^{\ell_M}$ and H_2: $\{0,1\}^{2\ell_M} \rightarrow \{0,1\}^{2\ell_M}$, a PTH function THF as $\mathsf{THF}(P, \Gamma, M) = \mathsf{H}_1(P, (\Gamma\|M) \oplus \mathsf{H}_2(\Gamma\|M))$.

The SS-MOW and SS-UD security of *Construction 1* under the (quantum) random oracle model follows directly from [32, §8], while SM-eTCR and SS-TOW analyses mirror those of SM-TCR and SM-PRE in [32, §8], differing only in the input arrangement of H_3.

Moreover, the security of Construction 2 is established through reductions to well-defined security notions of keyed hash (KH) functions in the (quantum) random oracle model. Specifically, the SM-eTCR of PTH is reduced to the SM-SPR of KH; the SS-MOW and SS-TOW of PTH are respectively reduced to the SS-OW of KH; and the SS-UD is directly reduced to the SS-UD of KH. We elaborate on the security analysis in Appendix B (in supplemental materials).

ACS **from** WOTS+CP**.** The core of our ACS lies in the variant WOTS+CP, which modifies WOTS+C [16] by replacing public tweaks with PRG-derived private tweaks. This modification ensures forward secrecy and provides secure authentication credentials (C) for HPoW. These private tweaks are authenticated through the WOTS+CP signature-chains, with chain tails verified by the Merkle tree. That is, each signature-chain is associated with a credential chain.

WOTS+CP uses a parameter κ to denote the length of messages, a Winternitz parameter w to define the length of each chain, with the number of chains denoted by $\ell = \left\lceil \frac{\kappa}{\log(w)} \right\rceil$. Rather than signing a message m, WOTS+CP signs the hash value $\mathsf{H}_4(m\|sa)$, where sa is a salt and H_4 is a keyed hash function. Two additional parameters, $S_{w,\kappa}$ and $\ell_0 \in \mathbb{N}$, ensure that the base-w representation $(a_1, \ldots, a_\ell)$ of $\mathsf{H}_4(m\|sa)$ satisfies the *compact properties*: i) $\sum_{i=1}^{\ell} a_i = S_{w,\kappa}$ and ii) $\forall i \in [\ell_0], a_i = 0$. The salt-finding probability [34] is $p_v = (\sum_{j=0}^{\ell}(-1)^j \cdot \tilde{C}(\ell, j)\tilde{C}(S_{w,\kappa} + \ell - jw - 1, \ell - 1))/(w^\ell 2^{\ell_0})$ where $\tilde{C}$ is the combination function. Eventually, the WOTS+CP scheme signs and verifies $\ell_1 = \ell - \ell_0$ signature-chains, each requiring $w-1$ calls to a PTH function THF (with parameter P) to generate chaining signature values.

Let $\mathsf{G} : \{0,1\} \rightarrow \{0,1\}^{2\kappa}$ be a pseudorandom generator. For each $y \in [\ell_1]$, we denote the initial signing-key element (SKE) and the corresponding credential seed (CS) of the y-th signature-chain by SKE^0_y and CS^0_y, respectively. For $z \in [w-1]$, subsequent SKEs in the y-th signature-chain are computed as $\mathsf{SKE}^z_y := \mathsf{THF}(P, \mathsf{C}^{w-z+1}_y, \mathsf{SKE}^{z-1}_y)$. Corresponding cre-

dentials are derived as $\{(\mathsf{CS}_y^z, \mathsf{C}_y^z) := \mathsf{G.Gen}(\mathsf{CS}_y^{z-1})\}_{z\in[w-1]}$. The credentials are used in reverse order of SKE growth to prevent recovering prior credentials from compromised seeds.

For each ACS instance, the $w\ell_1 - S_{w,\kappa}$ credentials associated with the ℓ_1 signature-chains are utilized to generate aliveness proofs. Therefore, we define an *epoch* as the period during which $n = w\ell_1 - S_{w,\kappa}$ aliveness proofs are produced based on a single ACS instance. To support long-term usage, multiple ACS instances are generated and linked sequentially using the integrated WOTS+CP. Specifically, the ACS_x for the x-th epoch is signed using the signing-key elements from the last ACS_{x-1}, alongside the generation of aliveness proofs in the $x-1$-th epoch. Once ACS_{x-1} is completed, the Merkle root of ACS_x is verified, allowing ACS_x to be used thereafter.

Here, we provide only a high-level overview of the principle behind ACS. Since ACS is tailored for SPAC, the formal definitions of its initialization, usage, and verification are deferred to the corresponding algorithms (Setup, ProofGen, and Verify) of SPAC.

D. Detailed Algorithms of SPAC

The detailed algorithms for SPAC are described below:

- $\mathsf{Setup}(1^\kappa, \mathsf{SetPms})$: A TTP runs this algorithm to first define the parameters $\mathsf{SetPms} = (\tilde{T}_{ps}, \tilde{T}_a, \tilde{T}_{ls}, \mathsf{aux})$. The auxiliary input $\mathsf{aux} = (\kappa, w, S_{w,\kappa}, \ell_0, \Delta_f, \tilde{T}_h^{av}, \tilde{T}_h^{wo}, \tilde{T}_{pc})$ consists of the parameters $\kappa, w, S_{w,\kappa}, \ell_0$ involved in the WOTS+CP scheme, a small failure-tolerant time Δ_f, the average-case and the worst-case runtimes $\tilde{T}_h^{av}$ and $\tilde{T}_h^{wo}$ of H (used by HP) on the client device respectively, and the average runtime $\tilde{T}_{pc}$ of the ProofGen algorithm excluding HPoW runtime.

Initialization of Parameters and Building Blocks. Then, the TTP initializes the building blocks including a PRG G as $pms_{\mathsf{G}} := \mathsf{G.Setup}(1^\kappa)$, a MT scheme M as $pms_{\mathsf{M}} := \mathsf{M.Setup}(1^\kappa)$, a hash-based proof of work HP as $pms_{\mathsf{HP}} := \mathsf{HP.Setup}(1^\kappa)$, and a tweakable hash function THF as $pms_{\mathsf{THF}} := \mathsf{THF.Setup}(1^\kappa)$. The difficulty d for running HP is derived from $\tilde{T}_a$, such that $O(T(d)) < \tilde{T}_a$. Moreover, the TTP initializes a KH function H_4 as $pms_{\mathsf{H}_4} := \mathsf{H}_4.\mathsf{Setup}(1^\kappa)$. For generating aliveness proofs, it chooses a uniformly random secret seed $\mathsf{ES}_0 \xleftarrow{\$} \mathcal{SK}_{\mathsf{PoPA}}$ to construct credential chains which provide authentication credentials to bootstrap the HPoW process. Let E denote the total number of epochs. Since ACS instances are self-replenishable, E can be variable and need only be sufficient to cover the entire $\tilde{T}_{ls}$.

ACS *Initialization.* The TTP will initialize two ACS instances to bootstrap the execution of SPAC in the first two epochs for the client, allowing it to independently replenish new ones. For any $x \in [2]$, the TTP computes the signature-chains and builds the Merkle Tree STr_x in the x-th epoch by iteratively performing the following steps: i) Compute the *epoch seed* (ES) ES_x (which will be used in the $(x+1)$-th epoch) and the *internal epoch seed* (IES) IES_x to be used by ACS_x (cf. Figure 13, and Algorithm 1), where $(\mathsf{ES}_x, \mathsf{IES}_x) := \mathsf{G.Gen}(\mathsf{ES}_{x-1})$; ii) Run $(\mathsf{CCS}_{x,0}, \mathsf{SKS}_{x,0}) := \mathsf{G.Gen}(\mathsf{IES}_x)$ to generate the initial *credential chain seed* (CCS) $\mathsf{CCS}_{x,0}$ and the initial *signing-key seed* (SKS) $\mathsf{SKS}_{x,0}$ using the x-th internal epoch seed IES_x; iii) Generate the rest of CCSs and the initial *credential seeds* (CS) as $\{(\mathsf{CCS}_{x,y}, \mathsf{CS}_{x,y}^0) := \mathsf{G.Gen}(\mathsf{CCS}_{x,y-1})\}_{y\in[\ell_1]}$; iv) Generate the rest of credentials in the ℓ_1 credential chains by literately calculating the CSs and the *credentials* (C) as $\{(\mathsf{CS}_{x,y}^z, \mathsf{C}_{x,y}^z) := \mathsf{G.Gen}(\mathsf{CS}_{x,y}^{z-1})\}_{y\in[\ell_1], z\in[w]}$, in which each $\mathsf{C}_{x,y}^z$ will be used as a private tweak; v) Compute the rest of SKSs $\{\mathsf{SKS}_{x,y}\}_{y\in[\ell_1]}$ and the initial *signing-key element* (SKE) $\{\mathsf{SKE}_{x,y}^0\}_{y\in[\ell_1]}$ for ℓ_1 signature-chains, where $(\mathsf{SKS}_{x,y}, \mathsf{SKE}_{x,y}^0) := \mathsf{G.Gen}(\mathsf{SKS}_{x,y-1})$; vi) For any $y \in [\ell_1]$ and $z \in [w-1]$, compute the z-th SKE in the y-th chain of the x-th epoch as $\mathsf{SKE}_{x,y}^z := \mathsf{THF}(P, \mathsf{C}_{x,y}^{w-z+1}, \mathsf{SKE}_{x,y}^{z-1})$; vii) Compute ℓ_1 leaves $\{\mathsf{lf}_{x,y} := \mathsf{THF}(P, \mathsf{C}_{x,y}^1, \mathsf{SKE}_{x,y}^{w-1})\}_{y\in[\ell_1]}$ to build the Merle tree $(\mathsf{STr}_x, \mathsf{st}_{\mathsf{BDS}}) := \mathsf{M.Build}(\{\mathsf{lf}_{x,y}\}_{y\in[\ell_1]})$ in the x-th epoch.

Algorithm 1 $\mathsf{SPAC.Setup}(1^\kappa, \mathsf{SetPms})$

Input: 1^κ, $\mathsf{SetPms} = (\tilde{T}_{ps}, \tilde{T}_a, \tilde{T}_{ls}, \mathsf{aux})$
Output: Pms
Parse $\mathsf{aux} = (\kappa, w, S_{w,\kappa}, \ell_0, \Delta_f, \tilde{T}_h^{av}, \tilde{T}_h^{wo}, \tilde{T}_{pc})$
$pms_{\mathsf{G}} := \mathsf{G.Setup}(1^\kappa)$; $pms_{\mathsf{M}} := \mathsf{M.Setup}(1^\kappa)$;
$pms_{\mathsf{HP}} := \mathsf{HP.Setup}(1^\kappa)$; $pms_{\mathsf{THF}} := \mathsf{THF.Setup}(1^\kappa)$
$pms_{\mathsf{H}_4} := \mathsf{H}_4.\mathsf{Setup}(1^\kappa)$
$\mathsf{ES}_0 \xleftarrow{\$} \mathcal{SK}_{\mathsf{PoPA}}$
$\ell_1 := w - \ell_0$ ▷ ℓ_1 is chosen to ensure a balanced Merkle tree
$n := w \cdot \ell_1 - S_{w,\kappa}$ ▷ The number of Aliveness proof of an ACS instance
$E := \lfloor \frac{\tilde{T}_{ls}}{n \cdot T_g} \rfloor$ ▷ T_g is the average time of generating an aliveness proof
for $x \in [2]$ **do**
  $(\mathsf{ES}_x, \mathsf{IES}_x) := \mathsf{G.Gen}(\mathsf{ES}_{x-1})$
  $(\mathsf{CCS}_{x,0}, \mathsf{SKS}_{x,0}) := \mathsf{G.Gen}(\mathsf{IES}_x)$
  for $y \in [\ell_1]$ **do**
    $(\mathsf{CCS}_{x,y}, \mathsf{CS}_{x,y}^0) := \mathsf{G.Gen}(\mathsf{CCS}_{x,y-1})$
    $(\mathsf{SKS}_{x,y}, \mathsf{SKE}_{x,y}^0) := \mathsf{G.Gen}(\mathsf{SKS}_{x,y-1})$
    for $z \in [w-1]$ **do**
      $\mathsf{SKE}_{x,y}^z := \mathsf{THF}(P, \mathsf{C}_{x,y}^{w-z+1}, \mathsf{SKE}_{x,y}^{z-1})$
    $\mathsf{lf}_{x,y} := \mathsf{THF}(P, \mathsf{C}_{x,y}^1, \mathsf{SKE}_{x,y}^{w-1})$
  $(\mathsf{STr}_x, \mathsf{st}_{\mathsf{BDS}}) := \mathsf{M.Build}(\{\mathsf{lf}_{x,y}\}_{y\in[\ell_1]})$
Find the first salt sa_2 meeting the compact properties of WOTS+CP for iC
$sk_{\mathsf{iC}} := \mathsf{ES}_1$; $\pi_{\mathsf{iC}} := (\mathsf{STr}_1.\mathsf{Rt}, \mathsf{STr}_2.\mathsf{Rt}, sa_2)$;
$\mathsf{Pms} := (\mathsf{SetPms}, pms_{\mathsf{G}}, pms_{\mathsf{M}}, pms_{\mathsf{HP}}, pms_{\mathsf{THF}}, pms_{\mathsf{H}_4}, \ell_1, E)$
Client's State Initialization:
- $i := 0$; $x := 1$; $y := 1$; $z := 0$
- $\mathbf{C} := \{\mathsf{C}_{x,y}^j\}_{j\in[w]}$; $\mathbf{SKE} := \{\mathsf{SKE}_{x,y}^v\}_{0\le v\le w-1}$; $T_{be}^i := \tilde{T}_{ps}$
- $\mathsf{st}_{\mathsf{iC}} := (i, x, y, z, \mathbf{C}, \mathbf{SKE}, \mathsf{STr}_{x+1}.\mathsf{Rt}, sa_{x+1}, \mathbf{m}, \mathsf{st}_{\mathsf{BDS}}, T_{be}^i)$.

Verifier's and Auditor's States Initialization:
- $\hat{i} := 0$; $\hat{x} := 0$; $\hat{y} := 0$; $\hat{\mathbf{z}} := (0, \ldots, 0)$, $\rho_{\hat{i}} := \emptyset$; $T_{ack}^{\hat{i}} := \tilde{T}_{ps}$
- $\mathsf{st}_{\mathsf{iA}} := \mathsf{st}_{\mathsf{iS}} = (\hat{i}, \hat{x}, \hat{y}, \hat{\mathbf{z}}, \rho_{\hat{i}}, T_{ack}^{\hat{i}})$

return $(sk_{\mathsf{iC}}, \mathsf{Pms}, \pi_{\mathsf{iC}})$

To sign $\mathsf{STr}_2.\mathsf{Rt}$, the TTP additionally finds the corresponding first valid salt sa_2 for iC (in advance) to save time. It sends the initial secret key $sk_{\mathsf{iC}} := \mathsf{ES}_1$ to iC via a secure channel (e.g., during the manufacturing process or using transport layer security protocols [41], [42]) and securely publishes the initial verification point $\pi_{\mathsf{iC}} := (\mathsf{STr}_1.\mathsf{Rt}, \mathsf{STr}_2.\mathsf{Rt}, sa_2)$ and parameters $\mathsf{Pms} := (\mathsf{SetPms}, pms_{\mathsf{G}}, pms_{\mathsf{M}}, pms_{\mathsf{HP}}, pms_{\mathsf{THF}}, pms_{\mathsf{H}_4}, d, \ell_1)$ to a bulletin board [43], [44].

Initialization of Participants' States. In the meantime, the client, the verifier, and the auditor will initialize their respective states. Let i ($\hat{i}$) denote the index of the last aliveness proof generated by the client iC (or verified by the verifier iS). x ($\hat{x}$) represents the index of the epoch to which the i-th ($\hat{i}$-th) aliveness proof belongs. y ($\hat{y}$) denotes the index of the

signature-chain in the x-th ($\hat{x}$-th) epoch. The client iC keeps a variable z to track the chain location of the signing-key element used for generating the i-th proof in the corresponding signature-chain. iC sets its initial state as $\mathsf{st}_{\mathsf{iC}} := (i,\ x,\ y,\ z,\ \mathsf{CCS}_{1,1}, \mathsf{SKS}_{1,1},\ \mathbf{C},\ \mathbf{SKE}, \mathsf{STr}_{x+1}.\mathsf{Rt},\ sa_{x+1},\ \mathbf{m},\ \mathsf{st}_{\mathsf{BDS}},\ T^i_{be})$, where $i := 0$, $x := 1$, $y := 1$, $z := 0$, $\mathbf{C} := \{\mathsf{C}^j_{x,y}\}_{j\in[w]}$, $\mathbf{SKE} := \{\mathsf{SKE}^v_{x,y}\}_{0\le v\le w-1}$, and $T^i_{be} := \tilde{T}_{ps}$. A set of credentials $\mathbf{C}$ and a set of signing-key elements $\mathbf{SKE}$ are cached for efficiency and forward secrecy (following the used-then-delete strategy). The two sets may cache more values when necessary, e.g., the credentials and signing-key elements of subsequent signature-chains. $\mathsf{st}_{\mathsf{iC}}$ may be dynamically updated and appended with other states along with the execution. To bootstrap the proof replenishment, iC records the Merkle root STr_{x+1} and its corresponding first valid salt sa_{x+1}. It also stores the signing message $\mathbf{m} := \mathsf{H}_4(\mathsf{STr}_{x+1}.\mathsf{Rt}\|sa_{x+1})$, which is mapped to ℓ_1 signature-chain locations $m_1, \ldots, m_{\ell_1} \in \{0, \ldots, w-1\}$. The client will compute the Merkle proof of the y-th leaf $\mathsf{lf}_{x,y} = \mathsf{THF}(P, \mathsf{C}^1_{x,y}, \mathsf{SKE}^{w-1}_{x,y})$ in STr_x as $\mathsf{pf}_{\mathsf{lf}_{x,y}} := \mathsf{M.GetPrf}(\mathsf{st}_{\mathsf{BDS}}, \mathsf{lf}_{x,y})$.

To verify the signature promptly, iS maintains a list of ℓ_1 values $\hat{\mathbf{z}} := (\hat{z}_1, \ldots, \hat{z}_{\ell_1})$, in which any $\hat{z}_j \in \{0, \ldots, w-1\}$ denotes the chain location of the last verified aliveness proof (i.e., the number of verified aliveness proofs) in the j-th chain of an epoch. The verifiers and the auditors initialize their states as $\mathsf{st}_{\mathsf{iA}} := \mathsf{st}_{\mathsf{iS}} = (\hat{i}, \hat{x}, \hat{y}, \hat{\mathbf{z}}, \rho_{\hat{i}}, T^{\hat{i}}_{ack})$, where $\hat{i} := 0$, $\hat{x} := 0$, $\hat{y} := 0$, $\hat{\mathbf{z}} := (0, \ldots, 0)$, $\rho_{\hat{i}} := \emptyset$, and $T^{\hat{i}}_{ack} := \tilde{T}_{ps}$.

- $\mathsf{ProofGen}(sk_{\mathsf{iC}}, \mathsf{st}_{\mathsf{iC}}, T_c)$: The client iC obtains the index i and the end-time T^i_{be} associated with the last aliveness proof from $\mathsf{st}_{\mathsf{iC}}$. It outputs a failure $\perp$ if the current time T_c meets the death condition: either $T_c < T^i_{be}$ or $T_c > T^i_{be} + \Delta_f$. Otherwise, iC sets $i := i + 1$ for generating the next aliveness proof.

The client retrieves three indices from $(x, y, z) \in \mathsf{st}_{\mathsf{iC}}$, to locate the credential for generating the next proof in an ACS instance. These include the current epoch index x, the signature-chain index y, the chain location z of the signing-key element linked to the $(i-1)$-th aliveness proof, and the y-th element $m_y \in \mathbf{m}$.

Credential Generation and Chain Switching Rules for ACS. In any credential chain, iC begins using the credential indexed from $z = 1$ up to $z = w - m_y$ (i.e., the location of the signature value determined by m_y). Once the credentials in a chain are exhausted, the client must switch to another chain. Based on y, z, and m_y, iC determines if any switch tasks (referred to as $\mathbf{ST}$) are needed to generate the i-th proof. When $0 \le z < w - m_y$, the required SKE is still located in the y-th signature-chain, then *no switch* task ($\mathbf{ST0}$) is required for the client. However, iC needs to do a *chain switch* ($\mathbf{ST1}$) in the x-th epoch to obtain the last SKE in the $(y+1)$-th chain when $z = w - m_y$ and $y \in [\ell_1 - 1]$. The proof generation requires iC to perform an *epoch switch* ($\mathbf{ST2}$) to get the last SKE in the first chain of the $(x+1)$-th epoch when $z = w - m_y$ and $y = \ell_1$. I.e., once the credentials in ACS_x are used up, iC switches to the next available ACS_{x+1}.

This algorithm (cf. Algorithm 2) is initially triggered with $\mathbf{ST0}$ at $z = 0$, and performs one of these switch tasks accordingly in each run to make preparations for the i-th proof as follows.

– $\mathbf{ST0}$: iC sets $z := z + 1$. Then, it gets the credential $\mathsf{C}^z_{x,y} \in \mathbf{C}$ and the signing-key element $\mathsf{SKE}^{w-z}_{x,y} \in \mathbf{SKE}$ from the corresponding sets in the state $\mathsf{st}_{\mathsf{iC}}$.

– $\mathbf{ST1}$: iC sets $y := y + 1$ and $z := 1$. Meanwhile, iC obtains the credential $\mathsf{C}^1_{x,y} \in \mathbf{C}$ and the last SKE of the new chain $\mathsf{SKE}^{w-1}_{x,y} \in \mathbf{SKE}$ from $\mathsf{st}_{\mathsf{iC}}$. If y is odd, it further computes leaf $\mathsf{lf}_{x,y} := \mathsf{THF}(P, \mathsf{C}^1_{x,y}, \mathsf{SKE}^{w-1}_{x,y})$ and gets its Merkle proof $\mathsf{pf}_{\mathsf{lf}_{x,y}} := \mathsf{M.GetPrf}(\mathsf{st}_{\mathsf{BDS}}, \mathsf{lf}_{x,y})$. For even y, the Merkle proof of $\mathsf{lf}_{x,y}$ is not needed, since it can verified by the corresponding leaf already included in the Merkle proof of its left sibling.

– $\mathbf{ST2}$: iC sets $x := x + 1$, $y := 1$, and $z := 1$. iC gets $\mathsf{C}^1_{x,y} \in \mathbf{C}$ and $\mathsf{SKE}^{w-1}_{x,y} \in \mathbf{SKE}$, and computes $\mathsf{pf}_{\mathsf{lf}_{x,y}}$ as in $\mathbf{ST1}$. iC also updates $\mathbf{m} := \mathsf{H}_4(\mathsf{STr}_{x+1}.\mathsf{Rt}||sa_{x+1})$.

Aliveness Proof Generations with Prepared Credentials. Afterwards, the client undertakes the following steps to generate the proof: i) Set the challenge-message $cm_i := \mathsf{C}^z_{x,y}$; ii) Generate the puzzle $p_i := \mathsf{HP.Gen}(d,\ cm_i)$; iii) Compute the solution $cnt_i := \mathsf{HP.Solve}(p_i)$; iv) Assemble the proof ρ_i according to one of the following *proof-cases* regarding $\mathbf{PfC0}$ (Inner Credential or Even Chain Switch), $\mathbf{PfC1}$ (Odd Chain Switch), and $\mathbf{PfC2}$ (Epoch Switch).

– $\mathbf{PfC0}$: if either $z \neq 1$ or ($y \bmod 2 = 0$ and $z = 1$), $\rho_i := (\mathsf{C}^z_{x,y}, \mathsf{SKE}^{w-z}_{x,y}, cnt_i)$;

– $\mathbf{PfC1}$: if $y \bmod 2 = 1$ and $z = 1$, $\rho_i := (\mathsf{C}^z_{x,y},\ \mathsf{SKE}^{w-z}_{x,y},\ \mathsf{pf}_{\mathsf{lf}_{x,y}},\ cnt_i)$;

– $\mathbf{PfC2}$: if $i \bmod n = 0$, $\rho_i := (\mathsf{C}^z_{x,y},\ \mathsf{SKE}^{w-z}_{x,y},\ cnt_i,\ \mathsf{STr}_{x+1}.\mathsf{Rt},\ sa_{x+1})$.

Algorithm 2 $\mathsf{SPAC.ProofGen}(sk_{\mathsf{iC}}, \mathsf{st}_{\mathsf{iC}}, T_c)$

Input : Secret key sk_{iC}, client's state $\mathsf{st}_{\mathsf{iC}}$, current time T_c
Output: Aliveness proof ρ_i or $\perp$
$i, T^i_{be} \leftarrow \mathsf{st}_{\mathsf{iC}}$
if $T_c < T^i_{be}$ ***or*** $T_c > T^i_{be} + \Delta_f$ **then** ▷ The death condition
  return $\perp$
$i := i + 1$
$x, y, z, m_y \leftarrow \mathsf{st}_{\mathsf{iC}}$
if $z < w - m_y$ **then** ▷ The required signing-key element is still located in the y-th signature-chain, and then it executes no switch task $\mathbf{ST0}$
  $z := z + 1$
  $\mathsf{C}^z_{x,y}, \mathsf{SKE}^{w-z}_{x,y} \leftarrow \mathsf{st}_{\mathsf{iC}}$
else if $z = w - m_y$ *and* $y \in [\ell_1 - 1]$ **then** ▷ It does a chain switch $\mathbf{ST1}$
  $y := y + 1$; $z := 1$
  $\mathsf{C}^1_{x,y}, \mathsf{SKE}^{w-1}_{x,y} \leftarrow \mathsf{st}_{\mathsf{iC}}$
  if y *is odd* **then**
    $\mathsf{lf}_{x,y} := \mathsf{THF}(P, \mathsf{C}^1_{x,y}, \mathsf{SKE}^{w-1}_{x,y})$
    $\mathsf{pf}_{\mathsf{lf}_{x,y}} := \mathsf{M.GetPrf}(\mathsf{st}_{\mathsf{BDS}}, \mathsf{lf}_{x,y})$
else if $z = w - m_y$ *and* $y = \ell_1$ **then** ▷ It does an epoch switch $\mathbf{ST2}$
  $x := x + 1$; $y := 1$; $z := 1$
  $\mathsf{C}^1_{x,y}, \mathsf{SKE}^{w-1}_{x,y} \leftarrow \mathsf{st}_{\mathsf{iC}}$
  $\mathsf{pf}_{\mathsf{lf}_{x,y}} := \mathsf{M.GetPrf}(\mathsf{st}_{\mathsf{BDS}}, \mathsf{lf}_{x,y})$; $\mathbf{m} := \mathsf{H}_4(\mathsf{STr}_{x+1}.\mathsf{Rt}||sa_{x+1})$
if $z \neq 1$ ***or*** ($y \bmod 2 = 0$ ***and*** $z = 1$) **then** ▷ $\mathbf{PfC0}$: Inner credential or even chain switch
  $\rho_i := (\mathsf{C}^z_{x,y}, \mathsf{SKE}^{w-z}_{x,y}, cnt_i)$
else if $y \bmod 2 = 1$ ***and*** $z = 1$ **then** ▷ $\mathbf{PfC1}$: Odd chain switch
  $\rho_i := (\mathsf{C}^z_{x,y}, \mathsf{SKE}^{w-z}_{x,y}, \mathsf{pf}_{\mathsf{lf}_{x,y}}, cnt_i)$
else if $i \bmod n = 0$ **then** ▷ $\mathbf{PfC1}$: Epoch switch
  $\rho_i := (\mathsf{C}^z_{x,y}, \mathsf{SKE}^{w-z}_{x,y}, cnt_i, \mathsf{STr}_{x+1}.\mathsf{Rt}, sa_{x+1})$
Run state update: run lines 9-22 of Algorithm 1 under the computation distribution rules
if $T'_c - T_c \ge \tilde{T}_a$ **then** ▷ Death
  return $\perp$
Gets a new current system time T'_c
$T^i_{be} := T'_c$
Discard $\mathsf{C}^z_{x,y}$ and $\mathsf{SKE}^{w-z}_{x,y}$ from $\mathsf{st}_{\mathsf{iC}}$
return ρ_i

State Update. During the generation of each aliveness proof, the client iC additionally performs two state update tasks: i) replenishing the new ACS_{x+2} before ACS_x is exhausted (since the client must know of the locations of the signature values in ACS_{x+1} in terms of the Merkle root of ACS_{x+2}); ii) generating and caching the credentials and signing-key elements (in the subsequent signature-chains) which are about to use later. These update tasks follow the procedure outlined in the Setup algorithm (details omitted for brevity). Simultaneously, iC updates the secret key sk_{iC} and various seeds in the state $\mathsf{st}_{\mathsf{iC}}$ as required by the update process. We assume that iC keeps secret key and seeds that are incapable of generating the values in $\mathbf{C}$ and $\mathbf{SKE}$. Whenever $\mathsf{STr}_{x+2}.\mathsf{Rt}$ becomes available, the client further finds the first valid salt sa_{x+2} such that the resulting signing message $\mathsf{H}_4(\mathsf{STr}_{x+2}.\mathsf{Rt}||sa_{x+2})$ satisfies the compact properties of WOTS+CP. We stress that all the above computations shall be evenly distributed across the proof generations within the current epoch, subject to the *condition* that $\mathsf{STr}_{x+2}.\mathsf{Rt}$ and sa_{x+2} are generated by the time the last aliveness proof of the x-th epoch is created.

After completing the state update tasks in each proof generation, iC gets a new current system time T_c' to check whether the time spent on generating the aliveness proof ρ_i is bounded by $\tilde{T}_a$. The client will output ρ_i for the client if $T_c' - T_c < \tilde{T}_a$, and $\perp$ otherwise. Finally, iC updates the end-time included in its state $\mathsf{st}_{\mathsf{iC}}$ with T_c', i.e., $T_{be}^i := T_c'$. The client will discard the credential $\mathsf{C}_{x,y}^z$ and the signing-key element $\mathsf{SKE}_{x,y}^{w-z}$ from its state and replace other values in $\mathsf{st}_{\mathsf{iC}}$ with the corresponding ones generated in the above process.

- $\mathsf{Verify}(\pi_{\mathsf{iC}}, \mathsf{st}_{\mathsf{iS}}, \rho_i, T_r)$: As the verifier iS receives an aliveness proof ρ_i, it obtains the index $\hat{i}$ and the check-time $T_{ack}^{\hat{i}}$ of the recently verified aliveness proof from its state $\mathsf{st}_{\mathsf{iS}}$. Then, this algorithm (cf. Algorithm 3) will output 0 if one of the following *aliveness-time* conditions is met: i) $T_r \leq T_{ack}^{\hat{i}}$; ii) $T_r - T_{ack}^{\hat{i}} > min(\tilde{T}_a, \tilde{T}_{pc} + cnt_i \cdot \tilde{T}_h^{wo}) + \Delta_f$; iii) $T_r - T_{ack}^i < cnt_i \cdot \tilde{T}_h^{av}$.[3] The satisfaction of the first condition implies that the value of T_r is invalid for verifying ρ_i, and the other two conditions mean that the aliveness is broken. The time $\tilde{T}_{pc} + cnt_i \cdot \tilde{T}_h^{wo}$ approximates the worst-case time for generating the i-th aliveness proof. The verifier further obtains the index $\hat{x}$ of the epoch, the index $\hat{y}$ of the corresponding signature-chain, the vector $\hat{\mathbf{z}} = (\hat{z}_1, \ldots, \hat{z}_{\ell_1})$, the signing-key element $\mathsf{SKE}_{\hat{x},\hat{y}}^{w-\hat{z}_{\hat{y}}}$ from the state $\mathsf{st}_{\mathsf{iS}}$.

We let $\mathbf{VR}\iota$ be the ι-th verification rule for checking the aliveness proof of the proof-case $\mathbf{PfC}\iota$. Specifically, iS checks the aliveness proof ρ_i as below.

– **VR0**: To verify $\rho_i = (\mathsf{C}_{x,y}^z, \mathsf{SKE}_{x,y}^{w-z}, cnt_i)$, iS gets potential verification points $\mathsf{SKE}_{\hat{x},\hat{y}}^{w-\hat{z}_{\hat{y}}}$ (from the last stored aliveness proof) and $\mathsf{lf}_{\hat{x},\hat{y}+1}$ (from the last stored Merkle proof). The algorithm initializes a verification indicator $vr := 0$, where $vr \neq 0$ means the corresponding verification step is valid. Since iS does not know from the proof whether an even chain switch event occurs, it checks the credential and signing-key element against both verification points, recording the valid one via vr. If $\mathsf{SKE}_{\hat{x},\hat{y}}^{w-\hat{z}_{\hat{y}}} = \mathsf{THF}(P, \mathsf{C}_{x,y}^z, \mathsf{SKE}_{x,y}^{w-z})$, iS sets $vr := 1$. Else if $vr = 0$ and $\mathsf{lf}_{\hat{x},\hat{y}+1} = \mathsf{THF}(P, \mathsf{C}_{x,y}^z, \mathsf{SKE}_{x,y}^{w-z})$, iS sets $vr := 2$. If $vr = 0$, the algorithm return 0. Then, iS sets $cm_i = \mathsf{C}_{x,y}^z$ and generates the HP puzzle $p_i = \mathsf{HP.Gen}(d, cm_i)$. The algorithm outputs 0 if the cnt_i included in ρ_i is not a valid solution to p_i (i.e., $\mathsf{HP.Verify}(p_i, cnt_i) = 0$). Finally, iS updates the states according the verification indicator vr. If $vr = 1$ (indicating the credential is within the current chain), iS sets $\hat{z}_{\hat{y}} := \hat{z}_{\hat{y}} + 1$. Otherwise, the signature-chain switches, and iS sets $\hat{y} := \hat{y} + 1$ and $\hat{z}_{\hat{y}} := 1$.

– **VR1**: Upon receiving $\rho_i = (\mathsf{C}_{x,y}^z, \mathsf{SKE}_{x,y}^{w-z}, \mathsf{pf}_{\mathsf{lf}_{x,y}}, cnt_i)$, iS computes the leaf $\mathsf{lf}_{x,y} = \mathsf{THF}(P, \mathsf{C}_{x,y}^z, \mathsf{SKE}_{x,y}^{w-z})$, and checks its Merkle proof. The algorithm outputs 0 if $\mathsf{M.Verify}(\mathsf{STr}_{\hat{x}}.\mathsf{Rt}, \mathsf{lf}_{x,y}, \mathsf{pf}_{\mathsf{lf}_{x,y}}) = 0$. Then, if $\hat{y} \neq 1$, iS sets $\hat{y} := \hat{y} + 1$. Moreover, iS sets $cm_i = \mathsf{C}_{x,y}^z$ and generates the HP puzzle $p_i = \mathsf{HP.Gen}(d, cm_i)$. The algorithm outputs 0 if cnt_i is not a valid solution to p_i. Otherwise, it sets the chain location as $\hat{z}_{\hat{y}} := 1$.

– **VR2**: For verifying $\rho_i = (\mathsf{C}_{x,y}^z, \mathsf{SKE}_{x,y}^{w-z}, cnt_i, \mathsf{STr}_{x+1}.\mathsf{Rt}, sa_{x+1})$, the algorithm first executes **VR0** to check

[3] We use the worst-case time bound to estimate the client's running time and avoid false negatives. Precise time modeling, as in [35], [45], is beyond this work's scope. We focus on the design's underlying constraints and requirements on time.

Algorithm 3 $\mathsf{SPAC.Verify}(\pi_{\mathsf{iC}}, \mathsf{st}_{\mathsf{iS}}, \rho_i, T_r)$

Input: Initial verification point π_{iC}, verifier's state $\mathsf{st}_{\mathsf{iS}}$, aliveness proof ρ_i, the receiving time T_r of the proof
Output: 0 if the proof is invalid, otherwise 1

$\hat{i}, T_{ack}^{\hat{i}}, \hat{x}, \hat{y}, \hat{\mathbf{z}} = (\hat{z}_1, \ldots, \hat{z}_{\ell_1}), \mathsf{SKE}_{\hat{x},\hat{y}}^{w-\hat{z}_{\hat{y}}}, \mathsf{lf}_{\hat{x},\hat{y}+1} \leftarrow \mathsf{st}_{\mathsf{iS}}$
if $T_r \leq T_{ack}^{\hat{i}}$ ***or*** $T_r - T_{ack}^{\hat{i}} > \min(\tilde{T}_a, \tilde{T}_{pc} + cnt_i \cdot \tilde{T}_h^{wo}) + \Delta_f$ ***or*** $T_r - T_{ack}^{\hat{i}} < cnt_i \cdot \tilde{T}_h^{av}$ **then** ▷ The value of T_r is invalid or the aliveness is broken
  return 0
begin VR0: verify ρ_i consisting of $\mathsf{C}_{x,y}^z, \mathsf{SKE}_{x,y}^{w-z}, cnt_i$
  $vr := 0$
  if $\mathsf{SKE}_{\hat{x},\hat{y}}^{w-\hat{z}_{\hat{y}}} = \mathsf{THF}(P, \mathsf{C}_{x,y}^z, \mathsf{SKE}_{x,y}^{w-z})$ **then**
    $vr := 1$ ▷ The credential is within the current signature-chain
  else if $vr = 0$ *and* $lf_{\hat{x},\hat{y}+1} = \mathsf{THF}(P, \mathsf{C}_{x,y}^z, \mathsf{SKE}_{x,y}^{w-z})$ **then**
    $vr := 2$ ▷ The credential is within the next signature-chain
  if $vr = 0$ **then**
    return 0
  $cm_i := \mathsf{C}_{x,y}^z$; $p_i := \mathsf{HP.Gen}(d, cm_i)$
  if $\mathsf{HP.Verify}(p_i, cnt_i) = 0$ **then**
    return 0 ▷ cnt_i is not a valid solution to p_i
  if $vr = 1$ **then**
    Update $\hat{z}_{\hat{y}} := \hat{z}_{\hat{y}} + 1$ in $\mathsf{st}_{\mathsf{iS}}$
  else
    Update $\hat{y} := \hat{y} + 1$ and $\hat{z}_{\hat{y}} := 1$ in $\mathsf{st}_{\mathsf{iS}}$
begin VR1 : verify ρ_i consisting of $\mathsf{C}_{x,y}^z, \mathsf{SKE}_{x,y}^{w-z}, \mathsf{pf}_{\mathsf{lf}_{x,y}}, cnt_i$
  Compute $\mathsf{lf}_{x,y} := \mathsf{THF}(P, \mathsf{C}_{x,y}^z, \mathsf{SKE}_{x,y}^{w-z})$
  if $\mathsf{M.Verify}(\mathsf{STr}_{\hat{x}}.Rt, lf_{x,y}, pf_{lf_{x,y}}) = 0$ **then**
    return 0 ▷ $\mathsf{pf}_{\mathsf{lf}_{x,y}}$ is not a valid Merkle proof for $\mathsf{lf}_{x,y}$
  if $\hat{y} \neq 1$ **then**
    $\hat{y} := \hat{y} + 1$
  $cm_i := \mathsf{C}_{x,y}^z$; $p_i := \mathsf{HP.Gen}(d, cm_i)$
  if $\mathsf{HP.Verify}(p_i, cnt_i) = 0$ **then**
    return 0 ▷ cnt_i is not a valid solution to p_i
  Update $\hat{z}_{\hat{y}} := 1$ in $\mathsf{st}_{\mathsf{iS}}$
begin VR2 : verify ρ_i consisting of $\mathsf{C}_{x,y}^z$, $\mathsf{SKE}_{x,y}^{w-z}$, cnt_i, $\mathsf{STr}_{x+1}.\mathsf{Rt}$, sa_{x+1}
  Execute **VR0**
  Compute $\mathbf{m} := \mathsf{H}_4(\mathsf{STr}_{x+1}.\mathsf{Rt}||sa_{x+1})$ and represent $\mathbf{m} = (m_1, \ldots, m_{\ell_1})$
  for $j \in [\ell_1]$ **do**
    if $\hat{z}_j - 1 \neq w - 1 - m_j$ **then**
      return 0 ▷ The signature is invalid
  Update $\hat{x} := \hat{x} + 1$; $\hat{y} := 1$; $\hat{\mathbf{z}} := (0, \ldots, 0)$ in $\mathsf{st}_{\mathsf{iS}}$
Update $i := \hat{i} + 1$ and $T_{ack}^i := T_r$ in $\mathsf{st}_{\mathsf{iS}}$; Append $(\rho_i, T_{ack}^i) \rightarrow \mathsf{st}_{\mathsf{iS}}$
return 1

$(\mathsf{C}^{z}_{x,y}, \mathsf{SKE}^{w-z}_{x,y}, cnt_i)$, and outputs 0 if $vr = 0$. Next, it verifies the signature of Merkle root STr_{x+1}.Rt of the new ACS instance stored in $\mathsf{st}_{\mathsf{iS}}$. iS computes the de-facto signing message $\mathsf{H}_4(\mathsf{STr}_{x+1}.\mathsf{Rt} \| sa_{x+1})$, and represents it as ℓ_1 based-w values $\mathbf{m} = (m_1, \ldots, m_{\ell_1})$ for verifying the corresponding signature $\sigma = (\sigma_1, \ldots, \sigma_{\ell_1})$. Note that for any $j \in [\ell_1]$, the signature value $\sigma_j = (\mathsf{C}^{w-m_j}_{x,j}, \mathsf{SKE}^{m_j}_{x,j})$ of σ associated with the chain location $\hat{z}_j = w - m_j$ has been verified so far along with the previous verifications of aliveness proofs. Therefore, iS does not need to iteratively apply THF to the credentials with the signing-key elements in each signature-chain to check the corresponding final signing-key element $\mathsf{SKE}^{w-1}_{x,j}$ for any $j \in [\ell_1]$ again. Thus, iS only needs to count the number of calls to THF that have been performed in each signature-chain and outputs 0 if $\hat{z}_j - 1 \neq w - 1 - m_j$ for any $j \in [\ell_1]$. Once the signature is accepted, iS updates the state: $\hat{x} := \hat{x} + 1$, $\hat{y} := 1$, and $\hat{\mathbf{z}} := (0, \ldots, 0)$.

If ρ_i passes the above verification, this algorithm sets $i := \hat{i} + 1$ and $T^i_{ack} := T_r$, appends (ρ_i, T^i_{ack}) to the state $\mathsf{st}_{\mathsf{iS}}$, and then outputs 1.[4]

- $\mathsf{Audit}(\pi_{\mathsf{iC}}, \mathsf{st}_{\mathsf{iS}}, \mathsf{st}_{\mathsf{iA}}, T)$: The auditor iA obtains the indices $\bar{i}$ and $\hat{i}$ of the last verified aliveness proof from its state $\mathsf{st}_{\mathsf{iA}}$ and the state $\mathsf{st}_{\mathsf{iS}}$ submitted by the verifier iS. Let $\mathsf{StateTrunc}(\mathsf{st}_{\mathsf{iS}}, j)$ be a function that truncates $\mathsf{st}_{\mathsf{iS}}$ to output a state $\mathsf{st}^j_{\mathsf{iS}}$ containing only the first $0 < j < \hat{i}$ state tuples, i.e., $\mathsf{st}^j_{\mathsf{iS}} = \{(v, \rho_v, T^v_{ack})\}_{v \in [j]}$. Then, iA utilizes the truncation function $\mathsf{StateTrunc}(\mathsf{st}_{\mathsf{iS}}, j)$ to output a substate of $\mathsf{st}_{\mathsf{iS}}$ which contains only the first j (where $0 < j < \hat{i}$) state tuples. This algorithm returns 0 if one of the following conditions is met: i) $T \leq T^{\hat{i}}_{ack}$; ii) $\bar{i} > \hat{i}$; iii) $\mathsf{st}_{\mathsf{iA}} \neq \mathsf{st}^{\bar{i}}_{\mathsf{iS}}$, where $\mathsf{st}^{\bar{i}}_{\mathsf{iS}} := \mathsf{StateTrunc}(\mathsf{st}_{\mathsf{iS}}, \bar{i})$; iv) and there exists a tuple $(\rho_j, T^j_r) \in \mathsf{st}_{\mathsf{iS}}$ for $\bar{i} < j \leq \hat{i}$ such that $\mathsf{Verify}(\pi_{\mathsf{iC}}, \mathsf{st}^{j-1}_{\mathsf{iS}}, \rho_j, T^j_r) = 0$. This algorithm sets $\mathsf{st}_{\mathsf{iA}} := \mathsf{st}_{\mathsf{iS}}$, and outputs 1.

Correctness. The aliveness proof is generated by an honest client operating normally and then verified by a verifier. This process repeats continuously. Specifically, the client starts to compute a new aliveness proof within a time window Δ_f after the last proof's end-time T_{be}. That is, for any time T in $(T_{be}, T_{be} + \Delta_f)$, the client iC with the state $\mathsf{st}^T_{\mathsf{iC}}$ (s.t., $T_{be} \in \mathsf{st}^T_{\mathsf{iC}}$) at T computes the next aliveness proof $\rho := \mathsf{ProofGen}(sk_{\mathsf{iC}}, \mathsf{st}^T_{\mathsf{iC}}, T)$. Note that the algorithms in the proof generation run within bounded time, e.g., the algorithms in the HPoW scheme which are bounded by a time related to the difficulty d. Thus, generating ρ takes a certain time upper-bounded by $\tilde{T}_a$ and lower-bounded by the puzzle-solving time (i.e., $cnt_i \cdot \tilde{T}^{av}_h$). This implied that the aliveness-time conditions hold when the verifier iS checks the receiving time of ρ. For the credential and the signing-key element included in ρ, the verification relies on the ACS which links signature-chains to Merkle tree leaves via THF and thus allows direct or indirect authentication via corresponding Merkle proofs. The completeness of HPoW ensures that the solution in ρ can be correctly verified by iS. Meanwhile, signing the Merkle root by the WOTS+CP scheme ensures the correctness in the authentication and verification of new Merkle roots. Therefore, ρ can pass the verification executed by the verifier and eventually be appended to the verifier's state. An honest verifier continuously appends valid aliveness proofs to its state, such that its state $\mathsf{st}^{T'}_{\mathsf{iS}}$ submitted to the auditor at time T' encompasses proofs included since the last audit. This indicates that proofs in the auditor's state at T' is encompassed by $\mathsf{st}^{T'}_{\mathsf{iS}}$. The auditor can thus truncate $\mathsf{st}^{T'}_{\mathsf{iS}}$ to check the validity of the update process of the state by verifying new proofs in sequence. The correctness of the verification guarantees that each new proof will be correctly verified. Finally, the last proof's check-time in $\mathsf{st}^{T'}_{\mathsf{iS}}$ is less than T'.

[4]The verification procedure efficiently prevents replay attacks. If no valid proof arrives within the latest time check window (defined at the beginning of this algorithm), the verifier raises an alarm and the administrator must fix the issue and reinitialize the scheme. Any proof submitted after this alarm is rejected, and any proof already verified and stored is ignored.

Security Analysis of SPAC. We briefly demonstrate the security of SPAC through the following theorems, with full proofs provided in Appendix C (in supplemental materials).

Theorem 1. *Let* G *be a* $(T, \epsilon^{\mathsf{IND}}_{\mathsf{PRG}})$*-secure* PRG*,* M *be a* $(T, \epsilon^{\mathsf{UF}}_{\mathsf{MT}})$*-secure* MT *scheme, and* HP *be a* $(d, T(d), \epsilon^{d}_{\mathsf{PoW}})$*-hard* HPoW *scheme. Let* THF *be a* $(T, \epsilon^{\mathsf{SS\text{-}TOW}}_{\mathsf{PTH}})$*-secure,* $(T, \epsilon^{\mathsf{SM\text{-}eTCR}}_{\mathsf{PTH}}, q)$*-secure,* $(T, \epsilon^{\mathsf{SS\text{-}UD}}_{\mathsf{PTH}})$*-secure, and* $(T, \epsilon^{\mathsf{SS\text{-}MOW}}_{\mathsf{PTH}})$*-secure* PTH *function family. Let* H_4 *be a* $(T, \epsilon^{\mathsf{MM\text{-}eTCR}}_{\mathsf{KH}}, q)$*-secure* KH *function family. We say that the maximum success probability over any polynomial-size adversary* $\mathcal{A}$ *running in time* $T^{\mathsf{PAlive}}_{\mathsf{PoPA}} \approx T$ *against the* PAlive *of* SPAC *is bound by* $\epsilon^{\mathsf{PAlive}}_{\mathsf{PoPA}} \leq \frac{E-1}{\eta} \cdot \epsilon^{\mathsf{MM\text{-}eTCR}}_{\mathsf{KH}} + \epsilon^{\mathsf{UF}}_{\mathsf{MT}} + \frac{E^2(\ell_1(w+1)+2)^2}{2} \cdot (E + \ell_1 + w) \cdot \epsilon^{\mathsf{IND}}_{\mathsf{PRG}} + \epsilon^{\mathsf{SM\text{-}eTCR}}_{\mathsf{PTH}} + Ew\ell_1\big((E+\ell_1+1) \cdot \epsilon^{\mathsf{IND}}_{\mathsf{PRG}} + w \cdot \epsilon^{\mathsf{SS\text{-}UD}}_{\mathsf{PTH}} + \epsilon^{\mathsf{SS\text{-}MOW}}_{\mathsf{PTH}}\big) + E(\ell_1(w+1)+2) \cdot \epsilon^{\mathsf{IND}}_{\mathsf{PRG}} + \epsilon^{d}_{\mathsf{PoW}}$*, where* p_v *is the probability of finding a hash value that satisfies the required properties of* WOTS+CP *with* q *queries and* $\eta = 1 - (1 - p_v)^q$.

We examine two adversary types, $\mathcal{A} = (\mathcal{F}, \mathcal{T})$, against the persistent aliveness (PAlive) property of SPAC. The forgery-adversary $\mathcal{F}$ aims to produce a malicious aliveness proof ρ^* for an uncompromised client. In contrast, the time-breaker $\mathcal{T}$ endeavors to outpace the challenger by generating an aliveness proof significantly faster, by leveraging a maliciously designed proof-generation circuit. Table II and Table III show the security proofs of SPAC's PAlive properties under $\mathcal{F}$ and $\mathcal{T}$, respectively.

TABLE II: Sequence of Games for PAlive-$\mathcal{F}$ of SPAC

Game	Description	Justification
Game 0	The real PAlive game	PAlive-Unforgeability
Game 1	As Game 0 but abort on signing-message hash collision	MM-eTCR of H_4
Game 2	As Game 1 but abort on forged Merkle proof	Unforgeability of MT
Game 3	As Game 2 but abort on PRG output collision	IND of PRG
Game 4	As Game 3 but abort on chaining-function collision	SM-eTCR of THF
Game 5	As Game 4 but abort on incorrect guess of the location indices (x^*, y^*, z^*) of the forgery	None
Game 6	As Game 5 but replace the guessed $\mathsf{SKE}^0_{x^*,y^*}$ and relevant secret seeds with random values	IND of PRG
Game 7	As Game 6 but replace $\{\mathsf{SKE}^j_{x^*,y^*}\}_{j \in [w-z^*]}$ with random values	SS-UD of THF
Game 8	As Game 7 but abort on finding valid preimage regarding $\mathsf{SKE}^{w-z^*}_{x^*,y^*}$	SS-MOW of THF

Malicious attempts of $\mathcal{F}$ must either rely on creating information inconsistent with honest client data or inferring the honest client's secret information before its usage. For instance, forging a proof may require $\mathcal{F}$ finding collisions of

H_4, breaking unforgeability of M, or violating the SM-eTCR or SS-MOW properties of THF to derive valid signing-key elements or credentials. To ensure these reductions, we replace values in target element generation with random ones, maintaining distribution consistency and using PRG security and SS-UD of THF. Finally, the determinism of HPoW ensures that its solution cannot be manipulated. For $\mathcal{T}$, the proof demonstrates that any malicious circuit designed to expedite proof generation cannot surpass the computational limits imposed by the hardness of PoW.

TABLE III: Sequence of Games for PAlive-$\mathcal{T}$ of SPAC

Game	Description	Justification
Game 0	The real PAlive game	PAlive-Time
Game 1	As Game 0 but replace all credentials and secret seeds with random values	IND of PRG
Game 2	As Game 1 but abort on a faster puzzle-solving	HARD of HP

Theorem 2. *With the same assumptions in Theorem 1, we say that the maximum success probability over any polynomial-size adversary $\mathcal{A}$ running in time $T^{\mathsf{Audit}}_{\mathsf{PoPA}} \approx T$ against the* Audit *of* SPAC *is bound by* $\epsilon^{\mathsf{Audit}}_{\mathsf{PoPA}} \leq \frac{E-1}{\eta} \cdot \epsilon^{\mathsf{MM\text{-}eTCR}}_{\mathsf{KH}} + \epsilon^{\mathsf{UF}}_{\mathsf{MT}} + \frac{E^2(\ell_1(w+1)+2)^2}{2} \cdot (E+\ell_1+w) \cdot \epsilon^{\mathsf{IND}}_{\mathsf{PRG}} + \epsilon^{\mathsf{SM\text{-}eTCR}}_{\mathsf{PTH}} + Ew\ell_1 \cdot \big((E+\ell_1+w+1) \cdot \epsilon^{\mathsf{IND}}_{\mathsf{PRG}} + \epsilon^{\mathsf{SS\text{-}TOW}}_{\mathsf{PTH}}\big)$.

The proof of this theorem is similar to that of Theorem 1 but shifts the focus to the SS-TOW property of THF, addressing cases where an adversary $\mathcal{A}$ forges a valid credential (tweak) of an honest client. In the Audit game, $\mathcal{A}$ aims to reconstruct historical states by deriving missing signing-key elements from existing ones in $\mathsf{st}_{\mathsf{iS}}$, provided the corresponding credentials are known. Thus, the attack primarily targets credentials rather than signing-key elements. We show the security proof of SPAC's Audit property under $\mathcal{A}$ in Table IV.

TABLE IV: Sequence of Games for Audit-$\mathcal{A}$ of SPAC

Game	Description	Justification
Game 0	The real Audit game	Audit
Game 1	As Game 0 but abort on signing-message hash collision	MM-eTCR of H_4
Game 2	As Game 1 but abort on forged Merkle proof	Unforgeability of MT
Game 3	As Game 2 but abort PRG output collision	IND of PRG
Game 4	As Game 3 but abort abort on chaining function collision	SM-eTCR of THF
Game 5	As Game 4 but abort on incorrect guess of the location indices (x^*, y^*, z^*) of the forgery	None
Game 6	As Game 5 but replace the guessed credential $\mathsf{C}^{z^*}_{x^*,y^*}$ and the relevant secret seeds with random values	IND of PRG
Game 7	As Game 6 but abort on finding a valid unused credential	SS-TOW of THF

VI. Performance Evaluation

Implementation Overview. To demonstrate the practicality of our proposal, we implement it on a Raspberry Pi 3 (simulating the client) and a PC with an AMD Ryzen 9 7945HX CPU and 16 GB RAM (acting as the verifier and auditor), using the MIRACL cryptographic library [46].[5] We consider 128-bit security against classical attacks for choosing the parameters of the underlying building blocks. Specifically, we instantiate the building blocks PRG with counter-mode AES [47] and hash functions with the SHA-2 family [48] (for both PTH and HPoW), while using the first PTH construction for efficiency. We benchmark the PoPA schemes across key metrics: system setup time, proof generation time, proof verification time, and audit time, averaging runtimes over 10,000 trials.

To replicate the real-world conditions, where the operation of the SPAC client program must minimize disruption to its core business processes (simulated by OpenPLC), we set the priority of the SPAC's client-test process to the lowest, allocating 1 CPU core and a maximum of 5% of CPU resources. Our PoPA design prioritizes client's performance. Since verification tasks of SPAC are parallelizable, verifier performance in a single client-verifier setup scales linearly with multiple clients, given sufficient computational resources.

Parameters Selection. We discuss parameter selection principles, which vary by application (e.g., smart grids, chemical plants) and devices. For WOTS+CP, we aim to minimize proof size by selecting a small number of signature-chains ℓ_1, which may increase the Winternitz parameter w. We also balance the Merkle tree by adjusting $\ell_0 = 0$. The parameters w and ℓ_1 ensure the checksum $S_{w,\kappa}$ is between 0 and $(w-1)\ell_1$. A suitable $S_{w,\kappa}$ is chosen to maximize available signing-key elements for aliveness proofs without significantly increasing computational overhead. In the following, we consider parameters $d \in \{8, 10, 12, 14, 16\}$, $w = 2^4$, $\ell_1 = 2^6$, $\ell_0 = 0$ and $S_{w,\kappa} = 2^9$. The HPoW difficulty can be configured based on the available computational resources to ensure that the worst-case runtime of ProofGen remains within $\tilde{T}_a$. In our experiment, we set $\tilde{T}_a \leq cnt^{wo}(d) \cdot \tilde{T}^{wo}_h + \tilde{T}_{pc} + \Delta_f$, where $cnt^{wo}(d)$ denotes the maximum number of HPoW solution in d. Further details on HPoW difficulty are in Appendix D (in supplemental materials). Other time parameters (Δ_f, $\tilde{T}^{av}_h$, $\tilde{T}^{wo}_h$, $\tilde{T}_{pc}$) can be accurately determined by system/network experts, as outlined in [35], [45]. Meanwhile, the output lengths of PRG and hash functions are determined by our security analysis in Appendix C, specifically accounting for the security loss factors introduced during reductions. Over long-term usage, these loss factors are primarily dominated by the number of epochs, E, which we set to $E = 2^{23}$ (considering an aliveness interval of $\tilde{T}_a = 0.04$ seconds) to ensure coverage for four years under a difficulty level of $d = 8$. Notably, a lower difficulty increases E. Thus, we set the output lengths of PRG and all hash functions (except for the one used by HPoW) to 256 bits and 384 bits, respectively. Since we only consider low difficulty levels for HPoW (which correspond to its security parameter), the output length of HPoW is set to 224 bits, following [12, Lemma 1]. Additionally, we estimate $\Delta_f = 0.2$ milliseconds (ms) in a local gigabit network with an average latency, and empirically measure average $\tilde{T}_{pc} = 1.1$ ms, and hash operation times $\tilde{T}^{av}_h = 3.8$ microseconds (μs) and $\tilde{T}^{wo}_h = 2\tilde{T}^{av}_h = 7.4$ μs.

Setup Time. Our initial measurement focuses on setup time, representing the duration required for generating the initial verification point and initializing the states of entities at the end of the setup algorithm. Notably, since SPAC does not

[5] Our source code is publicly available at https://github.com/LannisterArthur/SPAC.git.

require the initialization of all authentication credentials, it exhibits a constant and small initialization cost.

TABLE V: Runtime Evaluation of SPAC (in seconds (s))

	d=8	d=10	d=12	d=14	d=16
Setup	0.017468	0.017466	0.017473	0.017548	0.017477
ProofGen	0.028812	0.106018	0.406337	1.582025	6.337587
Verify	0.000049	0.000202	0.000776	0.003179	0.012335
Audit	0.012678	0.051319	0.198355	0.833877	3.201985

Proof Generation Time. We recall the performance of the BDS algorithm [33] in terms of THF evaluations. For a Merkle tree of height ℓ_h, the average and worst-case costs are $\ell_h - 2$ and $\frac{3}{2}(\ell_h - 3)$ THF operations, respectively, with up to $\frac{\ell_h}{2}$ nodes stored for traversal. For HP performance, the average puzzle-solving cost is $O(2^{d+1})$ determined by the difficulty d. Due to SHA-2's non-regularity, a fraction θ of solving processes may exceed 2^{d+1} hash operations.[6] Since SPAC supports replenishment, its costs include the expected $\frac{1}{p_v}$ hash evaluations [16] for finding the first valid salt, $(w\ell_1 + \ell_1 - 1)$ THF and $(w\ell_1 + 2\ell_1 + 2)$ PRG operations for building the Merkle tree and initializing WOTS+CP. These replenishment costs are distributed across the $n = w\ell_1 - S_{w,\kappa}$ authentication credentials in the current epoch, with each proof generation sharing Rep $= \frac{1/p_v \text{ H} + (w\ell_1+\ell_1-1) \text{ THF} + (w\ell_1+2\ell_1+2) \text{ PRG}}{n}$ replenishment operations, which is small. Benchmark results show that SPAC if lightweight for client devices capable of solving easy HPoW puzzles.

Verification and Audit Time. The verification runtime mainly includes the costs of verifying the Merkle proof, the HP puzzle, and the credential (involving 1 THF). Verifying the Merkle proof requires $\log \ell_1$ THF operations. The cost of HP.Verify is typically constant (1 H) with a large probability $1 - \theta$, and equals HP.Solve with probability θ. The audit runtime is approximately N' times the verification runtime, where N' is the number of proofs being audited, determined by the audit frequency. Here, we measure the runtime with $N' = 2^8$.

Communication and Storage Costs. We evaluate the communication cost by the size of the aliveness proof generated by a client, categorized into three cases. In **PfC0**, the proof consists of a 256-bit credential, 384-bit signing-key element, and 32-bit counter. **PfC1** adds a Merkle proof with $\log \ell_1$ THF values (6×384 bits), while **PfC2** includes a 384-bit Merkle root and a 32-bit salt. The total number of proofs in each case is $w\ell_1 - S_{w,\kappa} - \frac{\ell_1}{2} - 1$, $\frac{\ell_1}{2}$, and 1, respectively.

In summary, the proof sizes in SPAC are 84 bytes, 372 bytes, and 136 bytes, with ratios of 93.55%, 6.25%, and 0.2% for **PfC0**, **PfC1**, and **PfC2**, respectively.

The client's storage cost mainly encompasses two 256-bit PRG seeds, w cached credentials and w signing-key elements (i.e., $2^4 \times 256$ bits and $2^4 \times 384$ bits), a 384-bit Merkle root, and a 32-bit salt. For a Merkle tree of height 6, the BDS algorithm requires storing 14 tree nodes (14×384 bits). Thus, the client's total storage cost is about 2068 bytes.

Verifiers in CPS deployments are typically the most provisioned nodes, e.g., SCADA, plant masters, edge gateways, or cloud controllers. These nodes already ingest and retain high-rate time-series telemetry and event logs in dedicated storage tiers (process historians or time-series databases with compression, retention, and tiering). PoPA can simply reuse the same tier. The verifier's persistent state consists primarily of aliveness proofs. It may keep raw proofs only for a short retention window. After auditing, the verifier can either store compact commitments (e.g., per-window digests aggregated via a Merkle tree) for long-term history or delete the proofs entirely when policy permits.

[6] θ is about 13% obtained by our empirical experiments in $10,000$ repeated trials for difficulties ranging from 8 to 16.

TABLE VI: Performance Comparison

		XMSS + HPoW	SPAC
Proof Num		N	Unlimit
Computation	Setup	2N THF + N PRG	$2(w+1)\ell_1$-2 THF+ $2w\ell_1$ + $4\ell_1$+4 PRG + $1/p_v$ H
	ProofGen	1 BDS($\log N$) + 1 OTS-Sign + 1 HP-Solve	1 BDS($\log \ell_1$) + 1 PRG + 1 HP-Solve + 1 Rep
	Verify	$\log N$ THF + 1 HP-Solve + 1 OTS-Verify	$\log \ell_1$ THF + 1 HP-Solve + 1 THF
	Audit	N XMSS-Verify	N SPAC-Verify
Size	pk	2 THF	2 THF + 1 H
	sk	1 PRG-Seed	1 PRG-Seed
	Proof	1 OTS-Signaure + 1 MTPf($\log N$) +1 HP-Solution	1 PRG + 1 THF+ 1 MTPf($\log \ell_1$) + 1 HP-Solution
	$\mathsf{st}_{\mathsf{IC}}$	1 BDS ($\log N$)+ 1 OTS-State	1 BDS($\log \ell_1$) +4 PRG + $2w$ (THF + PRG)
	$\mathsf{st}_{\mathsf{IS}}$	N XMSS-Proof	N SPAC-Proof

Comparison. We compare our PoPA construction with a naive solution that combines a FSDS and HPoW, as discussed in Section II. Our goal is a fair comparison of schemes that can achieve similar properties like PAlive and Audit. For simplicity, we use an instantiation of FSDS with the standardized XMSS [14], based on a Merkle tree and OTS, similar to SPAC. Since SPAC requires only a one-time credential for authentication, it is more efficient than signing an HPoW solution in FSDS using OTS. Therefore, we do not further instantiate OTS. However, one could consider using an OTS scheme from the WOTS family.

We let "BDS(ℓ_h)" and "MTPf(ℓ_h)" respectively denote the costs of BDS and the size of Merkle tree proofs based on a Merkle tree with the height ℓ_h. Note that a higher Merkle tree implies the more costly BDS($\cdot$) and MTPf(ℓ_h). The replenishment cost of SPAC (as analyzed above) is denoted by "Rep", which only needs a few hash and PRG operations. We may also abuse the name of the function/algorithm to represent the costs (either in computation or size). The comparison results are summarized in Table VI in the worst case. The concrete costs of XMSS and WOTS+C can be found in [16], [49]. Roughly speaking, each WOTS+C singing operation may involve $S_{w,\kappa}$ THF, and $\frac{1}{p_v}$ H in expectation. Hence, SPAC is much more efficient than FSDSP from all perspectives. Moreover, SPAC provides unlimited aliveness proofs and very fast Setup and Verify algorithms.

VII. Conclusions

We have addressed the critical need for PoPA in resource-constrained environments by providing a formal security model and an innovative construction, SPAC. It integrates a new deterministic HPoW scheme and a forward-secure ACS, along with an OTS scheme, WOTS+CP, adapted from a novel PTH function, enabling unlimited aliveness proofs with minimal proof size. In summary, SPAC demonstrates the efficiency and effectiveness of lightweight PoPA with provable

security in the standard model. For future work, addressing the open questions outlined in Appendix D (in supplemental materials) could be intriguing.

REFERENCES

[1] J.-L. Roux and G. Ash, "Path Computation Element (PCE) Communication Protocol Generic Requirements," RFC 4657, Sep. 2006. [Online]. Available: https://www.rfc-editor.org/info/rfc4657

[2] C. Jin, Z. Yang, M. van Dijk, and J. Zhou, "Proof of aliveness," in *ACSAC*. ACM, 2019, pp. 1–16.

[3] Z. Yang, C. Jin, X. Cao, M. van Dijk, and J. Zhou, "Optimizing proof of aliveness in cyber-physical systems," *IEEE Trans. Dependable Secur. Comput.*, vol. 21, no. 4, pp. 3610–3628, 2024.

[4] M. Baker and P. Eavis, "Baltimore investigation turns to ship's deadly mechanical failure," https://www.nytimes.com/2024/03/30/us/baltimore-bridge-ship-mechanical.html, 2024, accessed: 2024-04-04.

[5] R. Naraine, "Russian hackers used ot attack to disrupt power in ukraine amid mass missile strikes," https://www.securityweek.com/russian-hackers-ot-attack-disrupted-power-in-ukraine-amid-mass-missile-strikes/, 2023, accessed: 2023-12-10.

[6] E. F. M. Josephlal, S. Adepu, Z. Yang, and J. Zhou, "Enabling isolation and recovery in plc redundancy framework of metro train systems," *Int. J. Inf. Secur.*, pp. 1–13, 2021.

[7] Z. Yang, Z. Bao, C. Jin, Z. Liu, and J. Zhou, "Plcrypto: A symmetric cryptographic library for programmable logic controllers," *IACR Trans. Symmetric Cryptol.*, vol. 2021, no. 3, pp. 170–217, 2021.

[8] J. Arquilla and M. Guzdial, "The solarwinds hack, and a grand challenge for CS education," *Commun. ACM*, vol. 64, no. 4, pp. 6–7, 2021.

[9] M. Alanazi, A. Mahmood, and M. J. M. Chowdhury, "SCADA vulnerabilities and attacks: A review of the state-of-the-art and open issues," *Comput. Secur.*, vol. 125, p. 103028, 2023.

[10] B. Wesolowski, "Efficient verifiable delay functions," *J. Cryptol.*, vol. 33, no. 4, pp. 2113–2147, 2020.

[11] G. Malavolta and S. A. K. Thyagarajan, "Homomorphic time-lock puzzles and applications," in *CRYPTO*, ser. Lecture Notes in Computer Science, vol. 11692. Springer, 2019, pp. 620–649.

[12] T. Jager and R. Kurek, "Short digital signatures and id-kems via truncation collision resistance," in *ASIACRYPT*. Springer, 2018, pp. 221–250.

[13] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Manubot, Tech. Rep., 2009.

[14] J. Buchmann, E. Dahmen, and A. Hülsing, "XMSS - A practical forward secure signature scheme based on minimal security assumptions," in *PQCrypto*, vol. 7071. Springer, 2011, pp. 117–129.

[15] A. Hülsing, C. Busold, and J. Buchmann, "Forward secure signatures on smart cards," in *SAC*, vol. 7707. Springer, 2012, pp. 66–80.

[16] A. Hülsing, M. A. Kudinov, E. Ronen, and E. Yogev, "SPHINCS+C: compressing SPHINCS+ with (almost) no cost," in *SP*. IEEE, 2023, pp. 1435–1453.

[17] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *CRYPTO*. Springer, 1992, pp. 139–147.

[18] E. Gabber, M. Jakobsson, Y. Matias, and A. J. Mayer, "Curbing junk e-mail via secure classification," in *FC*, vol. 1465. Springer, 1998, pp. 198–213.

[19] J. A. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," in *EUROCRYPT*. Springer, 2015, pp. 281–310.

[20] M. Ball, A. Rosen, M. Sabin, and P. N. Vasudevan, "Average-case fine-grained hardness," in *STOC*. ACM, 2017, pp. 483–496.

[21] ——, "Proofs of work from worst-case assumptions," in *CRYPTO*, vol. 10991. Springer, 2018, pp. 789–819.

[22] N. Haller, "The s/key one-time password system," 1995.

[23] D. Kogan, N. Manohar, and D. Boneh, "T/key: Second-factor authentication from secure hash chains," in *CCS*. ACM, 2017, pp. 983–999.

[24] L. Lamport, "Password authentication with insecure communication," *Commun. ACM*, vol. 24, no. 11, pp. 770–772, Nov. 1981.

[25] R. Anderson, "Two remarks on public key cryptology," University of Cambridge, Computer Laboratory, Tech. Rep., 2002.

[26] M. Bellare and S. K. Miner, "A forward-secure digital signature scheme," in *CRYPTO*, vol. 1666. Springer, 1999, pp. 431–448.

[27] G. Itkis and L. Reyzin, "Forward-secure signatures with optimal signing and verifying," in *CRYPTO*, vol. 2139. Springer, 2001, pp. 332–354.

[28] M. Abdalla, F. Benhamouda, and D. Pointcheval, "On the tightness of forward-secure signature reductions," *J. Cryptol.*, vol. 32, no. 1, pp. 84–150, 2019.

[29] F. T. Leighton and S. Micali, "Large provably fast and secure digital signature schemes based on secure hash functions," Jul. 11 1995, uS Patent 5,432,852.

[30] M. Bellare and T. Ristenpart, "Simulation without the artificial abort: Simplified proof and improved concrete security for waters' IBE scheme," in *EUROCRYPT*, vol. 5479. Springer, 2009, pp. 407–424.

[31] A. Hülsing, J. Rijneveld, and F. Song, "Mitigating multi-target attacks in hash-based signatures," in *PKC*, vol. 9614. Springer, 2016, pp. 387–416.

[32] A. Hülsing and M. A. Kudinov, "Recovering the tight security proof of sphincs$^+$," in *ASIACRYPT*, vol. 13794. Springer, 2022, pp. 3–33.

[33] J. Buchmann, E. Dahmen, and M. Schneider, "Merkle tree traversal revisited," in *PQCrypto*, vol. 5299. Springer, 2008, pp. 63–78.

[34] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, "The sphincs$^+$ signature framework," in *CCS*. ACM, 2019, pp. 2129–2146.

[35] B. van der Sanden, Y. Li, J. van den Aker, B. Akesson, T. Bijlsma, M. Hendriks, K. Triantafyllidis, J. Verriet, J. Voeten, and T. Basten, "Model-driven system-performance engineering for cyber-physical systems," in *EMSOFT*. ACM, 2021, pp. 11–22.

[36] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs," in *EUROCRYPT*. Springer, 2006, pp. 409–426.

[37] V. Shoup, "Sequences of games: A tool for taming complexity in security proofs," *IACR Cryptol. ePrint Arch.*, vol. 2004, 2004.

[38] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "{BootStomp}: On the security of bootloaders in mobile devices," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 781–798.

[39] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, "Swatt: Software-based attestation for embedded devices," in *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*. IEEE, 2004, pp. 272–282.

[40] M. Bellare and T. Kohno, "Hash function balance and its impact on birthday attacks," in *EUROCRYPT*, vol. 3027. Springer, 2004, pp. 401–418.

[41] T. Dierks and E. Rescorla, "The transport layer security (tls) protocol version 1.2," Tech. Rep., 2008.

[42] E. Rescorla, "The transport layer security (tls) protocol version 1.3," Tech. Rep., 2018.

[43] C. Culnane and S. A. Schneider, "A peered bulletin board for robust use in verifiable voting systems," in *CSF*. IEEE, 2014, pp. 169–183.

[44] A. R. Choudhuri, M. Green, A. Jain, G. Kaptchuk, and I. Miers, "Fairness in an unfair world: Fair multiparty computation from public bulletin boards," in *CCS*. ACM, 2017, pp. 719–728.

[45] C. Moreno and S. Fischmeister, "Accurate measurement of small execution times - getting around measurement errors," *IEEE Embed. Syst. Lett.*, vol. 9, no. 1, pp. 17–20, 2017.

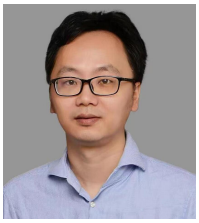
[46] "Miracl cryptographic library," 2018. [Online]. Available: https://bit.ly/2MltKVG

[47] E. B. Barker and J. M. Kelsey, *Recommendation for random number generation using deterministic random bit generators (revised)*. US Department of Commerce, Technology Administration, NIST, 2007.

[48] S. H. Standard, "Fips pub 180-2," *National Institute of Standards and Technology*, 2002.

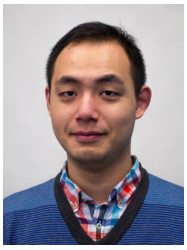
[49] P. Kampanakis and S. Fluhrer, "Lms vs xmss: Comparion of two hash-based signature standards," *Cryptology ePrint Archive*, 2017.

Xuelian Cao received her M.S. and Ph.D. degrees from the College of Computer and Information Science, Southwest University. She is currently a postdoctoral researcher at Tsinghua University. Her research interests include blockchain, applied cryptography, and fuzz testing.



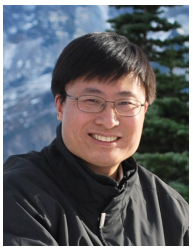
Zheng Yang received his Ph.D. degree from Horst Görtz Institute for IT Security, Ruhr-University Bochum, in 2013. He is currently a Professor with the College of Computer and Information Science, Southwest University, China. He was a post-doc researcher with the University of Helsinki, and the Singapore University of Technology and Design. His main research interests include information security, cryptography, and privacy.

Jianting Ning (Member, IEEE) received the Ph.D. degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, in 2016. He is currently a Professor with the Key Laboratory of Analytical Mathematics and Applications (Ministry of Education) and the Fujian Provincial Key Laboratory of Network Security and Cryptology, College of Computer and Cyber Security, Fujian Normal University, Fuzhou, China, and also the City University of Macau, Macau, China. Previously, he was a Research Scientist with the School of Computing and Information Systems, Singapore Management University, and a Research Fellow with the Department of Computer Science, National University of Singapore. He has published papers in major conferences/journals, such as ACM CCS, NDSS, ASIACRYPT, ESORICS, ACSAC, TIFS, and TDSC. His research interests include applied cryptography and information security.



Chenglu Jin received his Ph.D. degree from the Electrical and Computer Engineering Department, University of Connecticut, USA, in 2019. He is currently a tenure-track researcher in the Computer Security Group in Centrum Wiskunde & Informatica (CWI Amsterdam), the Netherlands. His research interests are cyber–physical system security, hardware security, and applied cryptography

Zhiming Liu 's area of research is software theory and methods. He is known for his work on Transformational Approach to Fault-Tolerant and Real-Time Systems Design and Analysis, Probabilistic Duration Calculus for System Dependability Analysis, Relational Semantics of Object-Oriented Programs, and rCOS Method for Model-Driven Development. His current interests focus on modelling, design and analysis of Human-Caber-Physical Systems which ever evolving hierarchical architecture of heterogeneous resources and components. Zhiming Liu studied Mathematics in university. He holds a MSc in Computing Science from the Software Institute of CAS (1988) and a PhD in Computer Science from the University of Warwick (1991). He joined Southwest University as a professor of computer science in 2016, and he heads the development of the Centre for Research and Innovation in Software Engineering (RISE). Before Southwest University, he had worked in three universities in the UK and the United Nations University – International Institute for Software Technology.



Jianying Zhou is a professor and center director for iTrust at Singapore University of Technology and Design (SUTD). He received PhD in Information Security from Royal Holloway, University of London. His research interests are in applied cryptography and network security, cyber-physical system security, mobile and wireless security. He is a co-founder & steering committee co-chair of ACNS. He is also steering committee chair of ACM AsiaCCS, and steering committee member of Asiacrypt. He has served 200+ times in international cyber security conference committees (ACM CCS & AsiaCCS, IEEE CSF, ESORICS, RAID, ACNS, Asiacrypt, FC, PKC etc.) as general chair, program chair, and PC member. He is associate editor-in-chief of IEEE Security & Privacy. He has also been in the editorial board of top cyber security journals including IEEE TDSC, IEEE TIFS, Computers & Security. He is an ACM Distinguished Member. He received the ESORICS Outstanding Contribution Award in 2020, in recognition of his contributions to the community.