Provably secure server-aided verification signatures

Wei Wu, Yi Mu, Willy Susilo, Xinyi Huang

Center for Computer and Information Security Research, School of Computer Science and Software Engineering, University of Wollongong, Wollongong, NSW 2522, Australia

Institute for Infocomm Research (I²R), Singapore

ARTICLE INFO

Article history:
Received 28 June 2010
Received in revised form 26 January 2011
Accepted 26 January 2011

Keywords:
Server-aided verification
Provably secure
Digital signatures

ABSTRACT

A server-aided verification signature scheme consists of a digital signature scheme and a server-aided verification protocol. With the server-aided verification protocol, some computational tasks for a signature verification are carried out by a server, which is generally untrusted; therefore, it is very useful for low-power computational devices. In this paper, we first define three security notions for server-aided verification signatures, i.e., existential unforgeability, security against collusion attacks and security against strong collusion attacks. The definition of existential unforgeability includes the existing security requirements in server-aided verification signatures. We then present, on the basis of existing signature schemes, two novel existentially unforgeable server-aided verification signature schemes. The existential unforgeability of our schemes can be formally proved both without the random oracle model and using the random oracle model. We also consider the security of server-aided verification signatures under collusion attacks and strong collusion attacks. For the first time, we formally define security models for capturing (strong) collusion attacks, and propose concrete server-aided verification signature schemes that are secure against such attacks.

1. Introduction

For power-constrained devices such as smart cards and mobile terminals, additional care must be taken with cryptographic algorithms due to the limited computational ability of those devices. Several techniques (e.g., pre-computation and off-line computation) have thus been introduced and adopted in order to improve the efficiency of cryptographic protocols. While such techniques can reduce the computational load, the computational requirement of many cryptographic systems (especially those with excellent security features) still remains too heavy for low-power devices. Pairing computation on elliptic curves is an example. Due to its elegant properties, pairing has been widely employed as a building block for lots of cryptographic schemes, in particular in the construction of identity-based encryption and short signatures. However, performing a pairing on an elliptic curve requires much more computational cost than executing an exponentiation computation. It remains a challenging task to reduce the computational cost in pairing-based cryptography.

A promising solution is to employ a powerful server assisting the low-power device (we refer to this as the client) to carry out cryptographic operations. This is known as “server-aided computation”. If the server is fully trusted, computations can be easily done through a secure channel between the client and the server. As an example, the client can send his/her private key to the server, which then can act on behalf of the client to decrypt ciphertexts or sign messages, and return the result to the client. However, the assumption of a fully trusted server seems too strong in practice, since clients are more likely to
face an untrusted server which could try to extract the secret of the client (in decryption or signing) or respond with a false result (in encryption or signature verification).

Many schemes for server-aided computation [1–12] have been proposed in the literature. A server-aided-verification signature scheme SAV-\(\Sigma\) consists of a digital signature scheme (\(\Sigma\)) and a server-aided verification protocol. Signatures can be verified by executing the server-aided verification protocol with the server, where the verification requires less computation than the original verification algorithm of \(\Sigma\). This notion was introduced by Quisquater and De Soete [13] for speeding up RSA verification with a small exponent. Lim and Lee [14] introduced this idea into discrete-logarithm-based schemes, by proposing efficient protocols for speeding up the verification of discrete-logarithm-based identity proofs and signatures. Their constructions are based on the “randomization” of the verification equation [14]. A different approach was introduced by Girault and Quisquater [15], which does not require pre-computation or randomization. Their server-aided verification protocol [15] is computationally secure, based on the hardness of a sub-problem of the underlying complexity problem in the original signature scheme. Hohenberger and Lysanskaya considered server-aided verification in the situation where the server is made up from two untrusted software packages, which are assumed not to communicate with each other [14]. Under this assumption, it allows a very light public computation task (typically one modular multiplication in the Schnorr scheme). Girault and Lefrance [16] proposed a more generalized model of server-aided verification without the assumption in [14]. A generic server-aided verification protocol for digital signatures from bilinear maps was also proposed [16]. Their protocol can be applied to signature schemes with similar constructions, such as the BB signature scheme [17] and the ZSS signature scheme [18].

Motivations and Contributions.

The motivations of this paper are desires to formally define the security of server-aided verification signatures and to construct new schemes that are secure under realistic security models. Girault and Lefrance [16] made the first attempt to define the security of server-aided verification signatures. Their definition consists of two aspects, namely the existential unforgeability of the signature scheme and the soundness of the server-aided verification protocol. The former notion is the same as that for digital signatures, and the latter requires that the server be unable to prove an invalid signature as valid using the server-aided verification protocol. Although this definition captures the essence of server-aided verification signatures, it is still worthwhile to define more elaborate models for further research on server-aided verification signatures. The contributions of this paper include three security models of server-aided verification signatures and concrete schemes secure under the new models. Our contributions are outlined as follows.

First, we introduce and define the existential unforgeability of server-aided verification signatures (or EUF-SAV-\(\Sigma\) for short). For server-aided verification signatures, we prove that EUF-SAV-\(\Sigma\) includes the existential unforgeability of signature schemes and the soundness of server-aided verification protocols, under the same assumption [16], that the server does not have any valid signature of the message when it tries to prove a signature of that message as valid. An existentially unforgeable server-aided verification signature scheme ensures that even the server (without colluding with the signer) is not able to forge a signature which can be proved to be valid by using the server-aided verification protocol.

Second, we consider the security of the ZSS signature [18] with the server-aided verification protocol proposed in [16]. The analysis shows that the server-aided verification ZSS in [16] can be made secure in our model, but requires more computational cost than that claimed in [16]. This is due to the difference between the security model defined in this paper and that in [16]. In our model, the server is allowed to execute the server-aided verification protocol with the verifier before proving to the verifier that an invalid signature is valid. This, however, is not allowed in [16] when making the security analysis of the server-aided verification ZSS. We believe that our model reflects a realistic case in real life.

Third, we introduce the server-aided verification for the Waters signature [19] and the BLS signature [20], respectively. We provide the first construction of SAV-Waters and SAV-BLS. SAV-Waters inherits the desirable property of the Waters signature, which can be proven to be existentially unforgeable without random oracles under the GBDH assumption. The existential unforgeability of SAV-BLS can be reduced to the hardness of the BDH problem in the random oracle model.

Last, we consider collusion between a signer and a server, who collaboratively prove an invalid signature to be valid. Such attacks were first sketched in [16] in the definition of “auxiliary non-repudiation”. Previous definitions (including EUF-SAV-\(\Sigma\)) are all based on the assumption that the malicious server does not have any valid signature of the message when it tries to prove an invalid signature of that message to be valid. For the first time, this paper formally defines security models for capturing the collusion attack and its stronger version in server-aided verification signatures, and proposes concrete server-aided verification signature schemes for protection against collusion attacks.

Paper Organization.

The rest of this paper is organized as follows. In Section 2, we define the notion of a server-aided verification signature scheme (SAV-\(\Sigma\)) whose existential unforgeability is defined in Section 3. We then analyze the existential unforgeability of a previously proposed SAV-\(\Sigma\) in Section 3. New constructions of existentially unforgeable SAV-\(\Sigma\) and the security analysis are given in Section 4. After that, we define the collusion and adaptive chosen message attacks along with a stronger version in Section 5. Concrete constructions of SAV-\(\Sigma\) secure against such attacks are also given in Section 5. Finally, we conclude this paper in Section 6.\(^1\)

\(^1\) The preliminary versions of two of the proposed protocols were published in [21] without security proofs.
2. Server-aided verification signatures

In this section, we provide the definitions of a signature scheme and a server-aided verification signature scheme.

2.1. Syntax of a signature scheme $\Sigma$

A signature scheme $\Sigma$ consists of the following algorithms:

- **Parameter-Generation**: $\text{ParamGen}(1^k) \rightarrow \text{param}$.

  This algorithm takes as input a security parameter $k$ and returns a string $\text{param}$, which denotes the common scheme parameters, including the description of the message space $\mathcal{M}$ and the signature space $\Omega$, etc. $\text{param}$ is shared among all the users in the system.

- **Key-Generation**: $\text{KeyGen}(\text{param}) \rightarrow (\text{sk}, \text{pk})$.

  This algorithm takes as input the system parameter $\text{param}$ and returns a secret/public key pair $(\text{sk}, \text{pk})$ for a user in the system.

- **Signature-Generation**: $\text{Sign}(\text{param}, m, \text{sk}, \text{pk}) = \sigma$.

  This algorithm takes as input the system parameter $\text{param}$, the message $m$ and the key pair $(\text{sk}, \text{pk})$, and returns a signature $\sigma$.

- **Signature-Verification**: $\text{Verify}(\text{param}, m, \sigma, \text{pk}) \rightarrow \{\text{Valid, Invalid}\}$.

  This algorithm takes as input the system parameter $\text{param}$, the message/signature pair $(m, \sigma)$ and the public key $\text{pk}$, and returns Valid or Invalid. $\sigma$ is said to be a valid signature of $m$ under $\text{pk}$ if $\text{Verify}(\text{param}, m, \sigma, \text{pk})$ outputs Valid. Otherwise, $\sigma$ is said to be invalid.

- **Completeness**: Any signature properly generated by $\text{Sign}$ can always pass through the verification in $\text{Verify}$. That is, $\text{Verify}(\text{param}, m, \text{Sign}(\text{param}, m, \text{sk}, \text{pk}), \text{pk}) = \text{Valid}$.

- **Existential unforgeability of $\Sigma$**: The standard notion of security for a signature scheme is called existential unforgeability under adaptive chosen message attacks [22], which is defined using the following game between a challenger $C$ and an adversary $A$.

  - **Setup**: The challenger $C$ runs the algorithms $\text{ParamGen}$ and $\text{KeyGen}$ to obtain system parameter $\text{param}$ and one key pair $(\text{sk}, \text{pk})$. The adversary $A$ is given $\text{param}$ and $\text{pk}$.

  - **Queries**: Proceeding adaptively, the adversary $A$ can request signatures of at most $q_s$ messages. For each sign query $m_i \in \{m_1, \ldots, m_{q_s}\}$, the challenger $C$ returns $\sigma_i = \text{Sign}(\text{param}, m_i, \text{sk}, \text{pk})$ as the response.

  - **Output**: Eventually, the adversary $A$ outputs a pair $(m^*, \sigma^*)$ and wins the game if:
    
    1. $m^* \notin \{m_1, \ldots, m_{q_s}\}$; and
    2. $\text{Verify}(\text{param}, m^*, \sigma^*, \text{pk}) = \text{Valid}$.

We define $\Sigma$ to be the probability that the adversary $A$ wins in the above game, taken over the coin tosses made by $A$ and the challenger.

**Definition 1.** A forger $A$ is said to $(t, q_s, \varepsilon)$-break a signature scheme $\Sigma$ if $A$ runs in time at most $t$, $A$ makes at most $q_s$ signature queries, and $\Sigma$ is a $t, q_s, \varepsilon$-existentially unforgeable against adaptive chosen message attacks if there exists no forger that $(t, q_s, \varepsilon)$-breaks it.

2.2. Syntax of a server-aided verification signature scheme SAV-$\Sigma$

A server-aided verification signature scheme SAV-$\Sigma$ consists of six algorithms: $\text{ParamGen}$, $\text{KeyGen}$, $\text{Sign}$, $\text{Verify}$, $\text{SA-Verifier-Setup}$, and $\text{SA-Verify}$. The first four algorithms are the same as those in an ordinary signature scheme $\Sigma$ defined in Section 2.1, and the last two are defined as follows:

- **Server-Aided-Verifier-Setup**: $\text{SA-Verifier-Setup}(\text{param}) \rightarrow \text{VString}$.

  This algorithm takes as input the system parameter $\text{param}$ and returns the bit string $\text{VString}$, which contains the information that can be pre-computed by the verifier. Note that $\text{VString}$ might be the same as $\text{param}$ if no pre-computation is required.

- **Server-Aided Verification**: $\text{SA-Verify}(\text{Server}(\text{param}), \text{Verifier}(m, \sigma, \text{pk}, \text{VString})) \rightarrow \{\text{Valid, Invalid}\}$.

$\text{SA-Verify}$ is an interactive protocol, between $\text{Server}$ and $\text{Verifier}$, who only has a limited computational ability and is not able to perform all computations in $\text{Verify}$ alone. Given the message/signature pair $(m, \sigma)$, as well as the public key $\text{pk}$ and the inner information $\text{VString}$, $\text{Verifier}$ checks the validity of $\sigma$ with the help of $\text{Server}$ by running $\text{SA-Verify}$. $\text{SA-Verify}$ returns Valid if $\text{Server}$ can convince $\text{Verifier}$ that $\sigma$ is valid. Otherwise, $\sigma$ is said to be invalid.
Completeness. There are two types of completeness in SAV-Σ:

1. **Completeness of Σ.** Any signature properly generated by \( \text{Sign} \) can always pass through the verification in \( \text{Verify} \). That is,
   \[
   \text{Verify}(\text{param}, m, \text{Sign}(\text{param}, m, sk, pk), pk) = \text{Valid}.
   \]

2. **Completeness of SAV.** An honest server can correctly convince the verifier about the validity of a signature. That is,
   \[
   \text{SA-Verify}(\text{Server}^{\text{param}}, \text{Verifier}(m, \sigma, pk, V\text{String})) = \text{Verify}(\text{param}, m, \sigma, pk).
   \]

2.3. **Computation-Saving in SAV-Σ**

Computation-Saving is probably the most obvious property that can distinguish a server-aided verification signature scheme SAV-Σ from an ordinary signature scheme Σ. This property enables the verifier in SAV-Σ to check the validity of signatures in a more computationally efficient way than that in Σ. This property is formally defined as follows.

**Definition 2 (Computation-Saving).** Let \( \Phi^{-\text{Verify}} \) and \( \Phi^{-\text{SA-Verify}} \) denote the verifier’s computational cost in \( \text{Verify} \) and \( \text{SA-Verify} \), respectively. A server-aided verification signature scheme SAV-Σ is said to be **Computation-Saving** if \( \Phi^{-\text{SA-Verify}} \) is strictly less than \( \Phi^{-\text{Verify}} \), i.e., \( \Phi^{-\text{SA-Verify}} < \Phi^{-\text{Verify}} \).

3. **Existentially unforgeable SAV-Σ**

It is clear that the security of SAV-Σ must include two security notions: existential unforgeability of \( Σ \) (EUF-Σ) and the soundness of SAV-Σ (Soundness-SA-Verify). The former is the same as that in Definition 1, while the latter is a new notion and only appears in the scenario of SAV-Σ. As usual, the soundness notion requires that the server should not be able to use \( \text{SA-Verify} \) to convince the verifier that an invalid signature is valid. The formal definition of the soundness depends on the assumption about the server. Below we will give the first security model of SAV-Σ under the same assumption as in [16]. We will define another model under different assumptions in Section 5.

3.1. **Definition of existential unforgeability of SAV-Σ**

Our first model follows the assumption in [16], namely, the server does not have the valid signature of the message when it tries to use \( \text{SA-Verify} \) to convince the verifier that an invalid signature of that message is valid. Under this assumption, it is not necessary to consider EUF-Σ and Soundness-SA-Verify separately. Instead, we will give a unified notion, called the existential unforgeability of SAV-Σ (or EUF-SAV-Σ for short), which implies EUF-Σ and Soundness-SA-Verify.

Briefly, EUF-SAV-Σ requires that the adversary should not be (computationally) capable of producing a signature of a new message which can be proved as Valid by \( \text{SA-Verify} \), even if the adversary acts as Server. A formal game-based definition is described as follows.

**Setup.** The challenger \( C \) runs the algorithms \( \text{ParamGen}, \text{KeyGen} \) and \( \text{SA-Verifier-Setup} \) to obtain system parameter \( \text{param} \), one key pair \( (sk, pk) \) and \( V\text{String} \). The adversary \( A \) is given \( \text{param} \) and \( pk \).**

**Queries.** The adversary \( A \) can make the following queries:

**Signature Queries.** Proceeding adaptively, the adversary \( A \) can request signatures of at most \( q_s \) messages. For each sign query \( m_i \in [m_1, \ldots, m_{q_s}] \), the challenger \( C \) returns \( \sigma_i = \text{Sign}(\text{param}, m_i, sk, pk) \) as the response.

**Server-Aided Verification Queries.** Proceeding adaptively, the adversary \( A \) can make at most \( q_v \) server-aided verification queries. For each query \( (m, \sigma) \), the challenger \( C \) responds by executing \( \text{SA-Verify} \) with the adversary \( A \), where the adversary \( A \) acts as Server and the challenger \( C \) acts as Verifier. At the end of each execution, the challenger returns the output of \( \text{SA-Verify} \) to the adversary \( A \).

**Output.** Eventually, the adversary \( A \) outputs a pair \( (m^∗, \sigma^∗) \) and wins the game if:

1. \( m^∗ \not\in \{m_1, \ldots, m_{q_s}\} \); and
2. \( \text{SA-Verify}(A^{(\text{param}, \text{InnerInfo})}, C^{(m^∗, \sigma^∗, pk, V\text{String})}) = \text{Valid} \), where \( \text{InnerInfo} \) refers to the inner information of \( A \) (e.g., the random element) in the generation of \( \sigma^∗ \).

We define SAV-Σ-Adv \( A \) to be the probability that the adversary \( A \) wins in the above game, taken over the coin tosses made by \( A \) and the challenger.

**Definition 3.** A forger \( A \) is said to \( (t, q_s, q_v, \varepsilon) \)-break a SAV-Σ if \( A \) runs in time at most \( t \), makes at most \( q_s \) signature queries and \( q_v \) server-aided verification queries, and SAV-Σ-Adv \( A \) is at least \( \varepsilon \). A SAV-Σ is \( (t, q_s, q_v, \varepsilon) \)-existentially unforgeable under adaptive chosen message attacks if there exists no forger that \( (t, q_s, q_v, \varepsilon) \)-breaks it.

When discussing security in the random oracle model, we add a fifth parameter \( q_h \) to denote an upper bound on the number of queries that the adversary makes to the random oracle.
Remarks on EUF-SAV-$\Sigma$. We note that in Setup, VString is not provided to the adversary who now is acting as Server. This is due to the concern that VString might contain some private information of the verifier, which must be kept secret in server-aided verification signatures. We can see that in the definition, adversary $A$ acts as the server and the challenger $C$ acts as the verifier. This will help $A$ to extract some information from VString. There is no need to consider the other case where $A$ acts as the verifier and $C$ acts as the (honest) server, as $A$ can perform all of the computations of an honest server.

We will show in Section 3.3 that the adversary defined in the above model is stronger than that in [16].

3.2. Further observations on EUF-SAV-$\Sigma$

We will show the relationships among EUF-SAV-$\Sigma$, EUF-$\Sigma$ and Soundness-SA-Verify.

It is self-evident that EUF-SAV-$\Sigma$ guarantees Soundness-SA-Verify. Otherwise, if there is an adversary that can prove that an invalid signature is valid by SA-Verify with success probability $\epsilon$, then it can also break the existential unforgeability of SAV-$\Sigma$ with the same probability. We now prove that EUF-SAV-$\Sigma$ also implies EUF-$\Sigma$.

Theorem 1. If SAV-$\Sigma$ is $(t, q_s, q_v, \epsilon)$-existentially unforgeable, then SAV-$\Sigma$ is $(t, q_s, \epsilon)$-existentially unforgeable.

Proof. Let the ordinary signature scheme be $\Sigma = (\text{ParamGen}, \text{KeyGen}, \text{Sign}, \text{Verify})$, and its server-aided verification counterpart be SAV-$\Sigma = (\Sigma, \text{SA-Verifier-Setup}, \text{SA-Verify})$. We prove the correctness of this theorem by converting a $(t, q_s, \epsilon)$ forgery $\Sigma_A$ to a $(t, q_s, 0, \epsilon)$ forgery SAV-$\Sigma_A$.

As defined in the game in Section 3.1, SAV-$\Sigma_A$ will obtain $(\text{param}, \text{pk})$ from its challenger of SAV-$\Sigma$. Then, SAV-$\Sigma_A$ acts as the challenger of $\Sigma_A$ as follows.

Setup. $(\text{param}, \text{pk})$ is given to $\Sigma_A$.

Queries. For each signature query $m_i$ from $\Sigma_A$, SAV-$\Sigma_A$ forwards $m_i$ to its challenger as a signature query of SAV-$\Sigma$. As defined, $\sigma_i = \text{Sign}(\text{param}, m_i, \text{sk}, \text{pk})$ will be returned as the answer. SAV-$\Sigma_A$ then forwards $\sigma_i$ to $\Sigma_A$. It is clear that each signature query from $\Sigma_A$ can be correctly answered.

Output. After making queries, $\Sigma_A$ will output a pair $(m^*, \sigma^*)$. SAV-$\Sigma_A$ sets $(m^*, \sigma^*)$ as its own output.

If $\Sigma_A((t, q_s, \epsilon))$-breaks the signature scheme $\Sigma$, then $m^*$ is not one of the signature queries and $\Pr[\text{Verify}(\text{param}, m^*, \sigma^*, \text{pk}) = \text{Valid}] \geq \epsilon$. Due to the completeness of SAV-$\Sigma$, if $\text{Verify}(\text{param}, m^*, \sigma^*, \text{pk}) = \text{Valid}$, then SAV-Verify will return Valid as well. Therefore, SAV-$\Sigma_A$ wins the game with the same probability $\epsilon$, without making any server-aided verification queries. This completes the proof. □

3.3. Analysis of the SAV-$\Sigma$ in Asiacrypt’05

In this section, we consider the existential unforgeability of the generic SAV-$\Sigma$ proposed by Girault and Lefranc [16]. Their server-aided verification protocol applies to signature schemes whose verification algorithms are similar to those in ZSS [18] and BB [17] signatures. We first review some fundamental background related to the protocol.

Bilinear Mapping: Let $G_1$ and $G_T$ be two groups of prime order $p$ and let $g$ be a generator of $G_1$. The map $e: G_1 \times G_1 \rightarrow G_T$ is said to be an admissible bilinear mapping if the following three conditions hold true:

- $e$ is bilinear, i.e., $e(g^a, g^b) = e(g, g)^{ab}$ for all $a, b \in \mathbb{Z}_p$.
- $e$ is non-degenerate, i.e., $e(g, g) \neq 1_{G_T}$.
- $e$ is efficiently computable.

We say that $(G_1, G_T)$ are bilinear groups if there exists the bilinear mapping $e: G_1 \times G_1 \rightarrow G_T$ as above, and $e$, and the group action in $G_1$ and $G_T$ can be computed efficiently. Such groups can be built from Weil pairing or Tate pairing on elliptic curves.

The Description of SAV-ZSS [16]

1. **ParamGen.** Let $(G_1, G_T)$ be bilinear groups where $|G_1| = |G_T| = p$, for some prime number $p \geq 2^k$, $k$ be the system security number and $g$ be the generator of $G_1$, $e$ denotes the bilinear map $G_1 \times G_1 \rightarrow G_T$. There is one cryptographic hash function $h: \{0, 1\}^* \rightarrow \mathbb{Z}_p$. The system parameter $\text{param} = (G_1, G_T, k, g, p, e, h)$.

2. **KeyGen.** The signer picks a random number $x \in \mathbb{Z}_p^*$ and keeps it as the secret key. The public key is set as $pk = g^x$.

3. **Sign.** For a message $m$ to be signed, the signer uses its secret key to generate the signature $\sigma = g^{m/x}$.

4. **Verify.** For a message/signature pair $(m, \sigma)$, everyone can check whether $e(\sigma, g^{h(m)} \cdot pk) = e(g, g)$. If the equation holds, output Valid. Otherwise, output Invalid.

5. **SA-Verifier-Setup.** Given the system parameter $\text{param} = (G_1, G_T, k, g, p, e, h)$, the verifier picks a random integer $t$ in $\mathbb{Z}_p$ and computes $K_1 = e(g, g)^t$. The VString is $(t, K_1)$.

6. **SA-Verify.** The verifier and the server interact with each other using the protocol described in Fig. 1.
Security of SAV-ZSS [16]. We now show that SAV-ZSS [16] is insecure in the model defined in Section 3.1, if the same \((t, K_1)\) is used in each execution of SAV-Verify described in Fig. 1.

We first briefly review the security conclusion of SAV-ZSS proved in [16]:

1. A malicious server is not able to convince a verifier that an invalid signature of a message \(m\) is valid by using SAV-Verify in Fig. 1, if:
2. The server does not know the ZSS signature of \(m\) and \(k\)-BCAA problem is hard (please refer to [16] for the definition of \(k\)-BCAA).

However, the malicious server considered in [16] is not allowed to execute SAV-Verify with the verifier, before it tries to prove to the verifier that an invalid signature is valid. We believe that this restriction is not reasonable, as the verifier in the real world would execute SAV-Verify with the server several times. In the model defined in Section 3.1, we allow the adversary (acting as the server) to choose any message–signature pair, and execute SAV-Verify with the challenger (acting as the verifier). This is analogous to the definition of existential unforgeability, where the forger is allowed to obtain valid signatures of messages chosen by itself. Under this model, SAV-ZSS [16] will be insecure if the same \((t, K_1)\) is used in SAV-Verify in Fig. 1. The following shows how the adversary in our model can break the existential unforgeability of SAV-ZSS [16]:

1. The adversary \(A\) first issues a signature query on a message \(m\). Let the response from the challenger be \(\sigma\).
2. \(A\) makes a server-aided verification request \((m, \sigma)\). As shown in SAV-Verify in Fig. 1, the challenger will send the adversary \(R = (g^{h(m)} \cdot pk)^t\).
3. \(A\) computes \(K_2 = e(\sigma, R)\). As \(\sigma\) is a valid ZSS signature of \(m\), \(K_1 = K_2 = e(g, g)^t\).
4. With the knowledge of \(K_1\), \(A\) is able to prove that any invalid signature is valid if the same \((t, K_1)\) is used in SAV-Verify. To do that, \(A\) just sends \(K_1\) to the challenger in every execution of SAV-Verify. Thus, \(A\) can always win the game defined in Section 3.1.

It is clear that the above attack will not work if the verifier pre-computes \(q_v + 1\) pairs \((t, K_1)\) in SAV-ZSS [16] and the adversary is allowed to make at most \(q_v\) server-aided verification queries. This will require more storage space for the verifier. Alternatively, the verifier can choose different \(t\), and compute \((g^{h(m)} \cdot pk)^t\) and \(e(g, g)^t\) in each execution of SAV-Verify. This however will lead to one more exponentiation in \(G_T\) than the computational cost of the verifier claimed in [16].

4. Existentially unforgeable SAV-BLS and SAV-Waters

This section describes new server-aided verification signature schemes: SAV-Waters and SAV-BLS, respectively.

4.1. Complexity assumptions

The bilinear mapping that we used in our protocol is the same as that defined in Section 3.3.

Bilinear Diffie–Hellman Problem (BDH). Given \((g, g^a, g^b, g^c)\) for some \(a, b, c \in \mathbb{Z}_p^*\), compute \(e(g, g)^{abc} \in \mathbb{G}_T\). An algorithm \(A\) has advantage \(\varepsilon\) in solving BDH on \((\mathbb{G}_1, \mathbb{G}_T)\) if

\[
\Pr[A(g, g^a, g^b, g^c) = e(g, g)^{abc} : a, b, c \in \mathbb{Z}_p^*] \geq \varepsilon.
\]

The probability is over the uniform random choice of \(a, b, c\) from \(\mathbb{Z}_p^*\), and over the coin tosses of \(A\).

---

1. SAV-ZSS in [16] is still secure against the adversary defined in [16]. However, the adversary in [16] is weaker than the one defined in this paper.
Verifier \((V\text{String}: (d, D))\) 

Server

<table>
<thead>
<tr>
<th>Input: ((m, \sigma = (\sigma_1, \sigma_2)), pk, \text{param})</th>
<th>(\sigma_1b \rightarrow K_1)</th>
<th>(K_2 = e(\sigma_2, V_0 \prod_{i \in M} V_i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Valid, if (K_1 = (PK \cdot K_2)^d)</td>
<td>Invalid, otherwise.</td>
<td></td>
</tr>
</tbody>
</table>

Note that the value \(D\) is precomputed, at each execution of \(SV\text{-Verify}\), verifier sends the same \(D\) to the server.

Fig. 2. The SAV protocol for the Waters signature.

| Table 1 | Comparisons I. |
|---|---|---|---|
| Verification | Pairing | Exponentiation on \(G_T\) | Multiplication on \(G_1\) |
| Waters | 2 | 0 | \(n\) |
| SAV-Waters | 1 | 1 | \(n\) |

### Bilinear Diffie–Hellman Assumption

The \((t, \varepsilon)\)-BDH assumption holds on \((G_1, G_T)\) if no \(t\)-time adversary has advantage at least \(\varepsilon\) in solving BDH on \((G_1, G_T)\).

### Decisional Bilinear Diffie–Hellman Oracle \(\Theta_{\text{BDH}}\) on \((G_1, G_T)\)

Given \((g, g^a, g^b, g^c, e(g, g)^d)\), this oracle outputs “1” if it is a BDH-tuple or “0” otherwise.

### Gap Bilinear Diffie–Hellman Problem (GBDH)

Given \((g, g^a, g^b, g^c)\) for some \(a, b, c \in \mathbb{Z}_p^*\), compute \(e(g, g)^{abc} \in G_T\) with the help of decisional bilinear Diffie–Hellman oracle \(\Theta_{\text{BDH}}\). An algorithm \(A\) has advantage \(\varepsilon\) in solving BDH on \((G_1, G_T)\) if

\[
\Pr[A(g, g^a, g^b, g^c, \Theta_{\text{BDH}}) = e(g, g)^{abc} : a, b, c \in \mathbb{Z}_p^*] \geq \varepsilon.
\]

The probability is over the uniform random choice of \(a, b, c\) from \(\mathbb{Z}_p^*\), and over the coin tosses of \(A\).

### Gap Bilinear Diffie–Hellman Assumption

The \((t, \varepsilon)\)-GBDH assumption holds on \((G_1, G_T)\) if no \(t\)-time adversary has advantage at least \(\varepsilon\) in solving GBDH on \((G_1, G_T)\).

### 4.2. Existentially unforgeable SAV-Waters

Our first protocol is based on the Waters signature [19].

1. \textbf{ParamGen}. Let \((G_1, G_T)\) be bilinear groups where \(|G_1| = |G_T| = p\), for some prime number \(p \geq 2^k\), \(k\) be the system security number and \(g\) be the generator of \(G_1\). \(e\) denotes the bilinear mapping \(G_1 \times G_1 \rightarrow G_T\). The system parameter \(\text{param} = (G_1, G_T, k, g, p, e)\). The message space \(M = \{0, 1\}^n\).

2. \textbf{KeyGen}. Given the system parameters \((G_1, G_T, k, g, p, e)\), the signer generates the public key \(pk\) as \((\vec{v}, PK, sk = x)\), where \(\vec{v}\) is a vector consisting of \(n + 1\) elements \(V_0, V_1, \ldots, V_n\) randomly selected in \(G_1\) and \(PK = e(g, g)^x\), where \(x\) is a random element in \(Z_p\).

3. \textbf{Sign}. For an \(n\)-bit message \(m \in \{0, 1\}^n\), let \(\mathcal{M} \subseteq \{1, 2, \ldots, n\}\) be the set of all \(i\) for which the \(i\)th bit of \(m\) is 1. The signature \(\sigma\) is constructed as \(\sigma = (\sigma_1, \sigma_2) = (g^r(V_0 \prod_{i \in M} V_i)^y, g^x)\), where \(r \in \mathbb{Z}_p\).

4. \textbf{Verify}. For a claimed signature \(\sigma = (\sigma_1, \sigma_2)\) of a message \(m\), this algorithm outputs Valid if and only if \(e(\sigma_1, g) = PK \cdot e(V_0 \prod_{i \in \mathcal{M}} V_i, \sigma_2)\). Otherwise, it outputs Invalid.

5. \textbf{SA-Verifier-Setup}. Given the system parameters \((G_1, G_T, k, g, p, e)\), the verifier randomly chooses \(d \in \mathbb{Z}_p^*\), then calculates \(D = g^d\). The VString is \((d, D)\).

6. \textbf{SA-Verify}. The verifier and the server interact with each other using the protocol described in Fig. 2.

### Computation-Saving

From Table 1, we replace the pairing operation \(e(\sigma_1, g)\) in the \(\text{Verify}\) algorithm with one exponentiation on \(G_T\). Thus, we have \(\Phi - \text{SA-Verify} < \Phi - \text{Verify}\).

### Theorem 2

The \textbf{SAV-Waters} described above is \((t, q_s, q_e, \varepsilon)\)-existentially unforgeable against chosen message attacks if the \((t + c_{(G_1, G_T)})(q_s + q_e + 1)\)-BDH assumption holds on \((G_1, G_T)\). Here, \(c_{(G_1, G_T)}\) is a constant that depends on \((G_1, G_T)\).
**Proof.** We will prove that if there is a \((t, q_1, q_2, q_3)\) adaptively chosen message such that adversary \(A\) wins the game defined in Section 3.1 with probability \(\varepsilon\), then there exists another algorithm \(B\) which can solve a random instance of the gap bilinear Diffie–Hellman problem in time \(t + c(G_1, G_T)(q_1 + q_2 + 1)\) with success probability \(\frac{c}{q_1q_2(n+1)}\). This contradicts the \((t + c(G_1, G_T)(q_1 + q_2 + 1), \frac{c}{q_1q_2(n+1)})\)-GDHD assumption on \((G_1, G_T)\).

Let \((G_1, G_T)\) be bilinear groups of prime order \(p\). Algorithm \(B\) is given \(g, g^a, g^b, g^c \in G_1\) which is a random instance of the GDHD problem. Its goal is to compute \(\epsilon(g, g^{abc})\). Algorithm \(B\) will simulate the challenger and interact with the adversary \(A\) as described below. \(A\) can adaptively make \textbf{Signature Queries} and \textbf{Server-Aided-Verification Queries}. To make the proof clearer, we introduce the notion of “special pair”. Let \(M_c\) be the set of \(A\)’s server-aided verification queries in the game. A message–signature pair \((m, \sigma) \in M_c\) is a special pair if:

1. by running \textbf{SA-Verify} \(A\) can convince the challenger that \((m, \sigma)\) is a valid message–signature pair; and
2. \(m\) has not appeared as one of the signature queries when \(A\) makes the server-aided verification query \((m, \sigma)\).

We now define the following two events:

- **E1:** There is a special pair in \(M_c\).
- **E2:** There is no special pair in \(M_c\).

It is clear that either \(E1\) or \(E2\) happens in the game, and thus \(\Pr[E1] + \Pr[E2] = 1\).

The Event \(\text{Succ}|E1\). If the event \(E1\) happens, then there is a special pair in \(A\)’s server-aided-verification queries. \(B\) picks a random integer \(j \in [1, 2, \ldots, q_3]\), and guesses this is the index of the first special pair.

1. **Setup.** \(B\) first sets an integer \(z = 4q_2\), and chooses an integer \(k \in \mathbb{Z}\). Then \(B\) chooses random \((n + 1)\)-length vectors \(\vec{a} = (\alpha_i)\) and \(\vec{b} = (\beta_i)\), where \(\alpha_i \in \mathbb{Z}_p\) and \(\beta_i \in \mathbb{Z}_p\), respectively.

   Meanwhile, for a message \(m \in \{0, 1\}^n\), we set \(M = \{1, 2, \ldots, n\}\) as the set of all \(i\) for which the \(i\)th bit of \(m\) is 1. Then we define three functions:

   \[ F(m) = (p - zk) + \alpha_0 + \sum_{i \in M} \alpha_i; \]
   \[ J(m) = \beta_0 + \sum_{i \in M} \beta_i; \]
   \[ K(m) = \begin{cases} 0, & \text{if } \alpha_0 + \sum_{i \in M} \alpha_i \equiv 0 \text{(mod } z), \\ 1, & \text{otherwise}. \end{cases} \]

\(B\) sets \(g_1 = g^a, g_2 = g^b\) and \(D = g^c\) where \(g^a, g^b\) and \(g^c\) are the inputs of the GBDH problem. \(B\) then calculates \(V_0 = g_1^{-kz + \alpha_0} g_0^\beta_0, V_i = g_0^\langle \alpha_i \rangle g_0^\beta_i, i = 1, 2, \ldots, n, PK = e(g_1, g_2),\) and sets \(\vec{v} = (V_i),\) the public key \(pk = (\vec{v}, PK)\).

2. **Signature Queries.** At any time the adversary \(A\) can request the signature of the input \(m\).

   (a) If \(K(m) = 0\), \(B\) terminates the simulation and reports failure.

   (b) Otherwise, \(K(m) \neq 0\) which implies \(F(m) \neq 0\), \(B\) chooses \(r \in \mathbb{Z}_p\), and generates the signature as

   \[ \sigma = (\sigma_1, \sigma_2) = \left( g_2^{-J(m)F(m)} V_0 \prod_{i \in M} V_i \right)^r; g_2^{-\overline{F(m)} g^r}. \]

   \(\sigma\) is a valid signature as shown below.

   Let \(\tilde{r} = r - \frac{b}{F(m)}\); we have

   \[ \sigma_1 = g_2^{-J(m)F(m)} V_0 \prod_{i \in M} V_i \]
   \[ = g_2^{-J(m)F(m)} (g_1^{F(m)} g^r)^{\tilde{r}} \]
   \[ = g_2^{ab} g^{-ab} g_2^{-J(m)F(m)} g_1^{F(m)} g^r \]
   \[ = g_2^{ab} g_1^{F(m)} g^r \]
   \[ = g^{ab} (g_1^{F(m)} g^r)^{\tilde{r}} \]
   \[ = g^{ab} \left( V_0 \prod_{i \in M} V_i \right)^{\tilde{r}}. \]

   And \(\sigma_2 = g_2^{-\overline{F(m)} g^r} = g^{r - \frac{b}{F(m)}} = g^\tilde{r}.\)
3. Server-Aided-Verification Queries. At any time the adversary $\mathcal{A}$ can make a Server-Aided-Verification query of $(m_l, \sigma_l)$, where $\sigma_l = (\sigma_{1l}, \sigma_{2l})$.

(a) For $l \in \{1, 2, \ldots, q_i\}$ and $l < j$:
- If $m_l$ has not appeared as one of the Signature Queries, $\mathcal{B}$ will execute the server-aided-verification protocol with $\mathcal{A}$ but output Invalid at the end of the protocol, no matter what $\mathcal{A}$'s response is in the protocol. It is clear that if $\mathcal{B}$'s guess of a special pair is correct, then $\mathcal{B}$'s output will be correct as well.
- Otherwise, $\mathcal{A}$ has made a signature query of $m_l$; let $\mathcal{B}$'s answer be $\sigma'_l = (\sigma'_{1l}, \sigma'_{2l})$. In this case, $\mathcal{B}$ will execute the protocol with $\mathcal{A}$. First, $\mathcal{B}$ sends $(\sigma_{1l}, D)$ to $\mathcal{A}$. Then, $\mathcal{A}$ responds with $K_{1l}$. After that $\mathcal{B}$ computes $\theta = \frac{\sigma_{1l}}{\sigma'_{2l}}$, $\vartheta = V_0 \prod_{i \in M} V_i \lambda = \frac{K_B}{e(\sigma_{1l}, D)}$, and issues the query $(g, \theta, \vartheta, D, \lambda)$ to $\mathcal{O}_{\text{GBDH}}$. Finally, $\mathcal{B}$ will output Valid if $\mathcal{O}_{\text{GBDH}}$ returns 1. Otherwise, the output is Invalid.

Note that if $\mathcal{O}_{\text{GBDH}}$ returns 1, i.e. $\lambda = e(\theta, \vartheta)^c$, then we have
\[
\frac{K_{1l}}{e(\sigma_{1l}, D)} = e\left(\frac{\sigma_{1l}}{\sigma'_{2l}}, V_0 \prod_{i \in M} V_i\right)^c
\]
\[
K_{1l} = e(\sigma'_{1l}, D) \cdot e\left(\frac{\sigma_{1l}}{\sigma'_{2l}}, V_0 \prod_{i \in M} V_i\right)^c
\]
\[
= e(g, g)^{abc} \cdot e\left(\frac{\sigma_{1l}}{\sigma'_{2l}}, V_0 \prod_{i \in M} V_i\right)^c \cdot e\left(\frac{\sigma_{1l}}{\sigma'_{2l}}, V_0 \prod_{i \in M} V_i\right)^c
\]
\[
= \left[PK \cdot e\left(\frac{\sigma_{1l}}{\sigma'_{2l}}, V_0 \prod_{i \in M} V_i\right)^c\right]^c,
\]
which means $\sigma_l$ is valid according to the protocol described in Fig. 2. Otherwise $\mathcal{O}_{\text{GBDH}}$ returns 0, and $\sigma_l$ is invalid.

(b) For $l \in \{1, 2, \ldots, q_i\}$ and $l = j$, let $(m^*, \sigma^*)$ be the $l$th query where $\sigma^* = (\sigma_{1*}, \sigma_{2*})$. After receiving $D$ from $\mathcal{B}$, $\mathcal{A}$ will send $K_{1l}$ to $\mathcal{B}$. As $(m^*, \sigma^*)$ is a special pair, $\mathcal{A}$ can prove it as valid by using the protocol $\text{SA-Verify}$ described in Fig. 2. $\mathcal{B}$ will receive $K_{1l}$ such that
\[
K_{1l} = (PK \cdot K_2^*)^c = e(g, g)^{abc} e\left(\sigma_{1l}, V_0 \prod_{i \in M} V_i\right)^c
\]
\[
= e(g, g)^{abc} e\left((g_1^{(m^*)} g^{(m^*)})^e, D\right).
\]
If $\alpha_0 + \sum_{i \in M^*} \alpha_i = k z$, then $K_{1l} = e(g, g)^{abc} e((\sigma_{1l})^{(m^*)}, D)$. $\mathcal{B}$ can solve the GBDH problem by computing $e(g, g)^{abc} = K_{1l}^c / e((\sigma_{1l})^{(m^*)}, D)$ and terminate the simulation.

$\mathcal{B}$ can output $e(g, g)^{abc}$ if and only if
(a) $\mathcal{B}$ does not abort during Signature Queries; the probability that this event happens is $\Pr[\bigwedge_{i=1}^{q_i} K(m_i) = 1]$.
(b) $\mathcal{B}$ makes a correct guess of a special pair, which happens with the probability $\frac{1}{q_i}$. This also guarantees that before $\mathcal{B}$ terminates the simulation, all server-aided-verification queries from $\mathcal{A}$ can be correctly answered.
(c) The above two events happen and $\alpha_0 + \sum_{i \in M^*} \alpha_i = k z$ in the special pair $(m^*, \sigma^*)$.
Therefore, the probability that $\mathcal{B}$ can successfully output $e(g, g)^{abc}$ is $\frac{1}{q_i} \Pr[\bigwedge_{i=1}^{q_i} K(m_i) = 1 \wedge \alpha_0 + \sum_{i \in M^*} \alpha_i = k z]$. We have
\[
\Pr\left[\bigwedge_{i=1}^{q_i} K(m_i) = 1 \wedge \alpha_0 + \sum_{i \in M^*} \alpha_i = k z\right] = \left(1 - \Pr\left[\bigvee_{i=1}^{q_i} K(m_i) = 0\right]\right) \Pr\left[\alpha_0 + \sum_{i \in M^*} \alpha_i = k z \bigg| \bigwedge_{i=1}^{q_i} K(m_i) = 1\right]
\]
\[
\geq \left(1 - \sum_{i=1}^{q_i} \Pr[K(m_i) = 0]\right) \Pr\left[\alpha_0 + \sum_{i \in M^*} \alpha_i = k z \big| \bigwedge_{i=1}^{q_i} K(m_i) = 1\right]
\]
\[
= \left(1 - \frac{q_i}{z}\right) \Pr\left[\alpha_0 + \sum_{i \in M^*} \alpha_i = k z \big| \bigwedge_{i=1}^{q_i} K(m_i) = 1\right]
\]
\[
= \left(1 - \frac{q_i}{z}\right) \frac{1}{n+1} \Pr\left[\bigvee_{i=1}^{q_i} K(m_i) = 0\right] \Pr\left[\bigwedge_{i=1}^{q_i} K(m_i) = 1\right]
\]
\[
= \left(1 - \frac{q_i}{z}\right) \frac{1}{n+1} \Pr\left[\bigwedge_{i=1}^{q_i} K(m_i) = 1\right] \Pr\left[\bigwedge_{i=1}^{q_i} K(m_i) = 1\right] \Pr\left[K(m^*) = 0\right]
\]
\[
\begin{align*}
&\geq \left(1 - \frac{q_i}{z}\right) \frac{1}{n+1} \frac{1}{z} \Pr\left[\bigwedge_{i=1}^{q_i} K(m_i) = 1 | (m^*) = 0\right] \\
&= \left(1 - \frac{q_i}{z}\right) \frac{1}{z(n+1)} \left(1 - \Pr\left[\bigwedge_{i=1}^{q_i} K(m_i) = 0 | (m^*) = 0\right]\right) \\
&\geq \left(1 - \frac{q_i}{z}\right) \frac{1}{z(n+1)} \left(1 - \sum_{i=1}^{q_i} \Pr[K(m_i) = 0 | (m^*) = 0]\right) \\
&= \left(1 - \frac{q_i}{z}\right) \frac{1}{z(n+1)} \left(1 - \frac{q_i}{z}\right) \\
&= \frac{1}{8q_i(n+1)}.
\end{align*}
\]

Therefore, \(\Pr[\text{Succ}\{E1\}] \geq \frac{1}{8q_i(n+1)}\).

**The Event Succ\{E2\}**. If the event \(E2\) happens, then there is no special pair in \(A\)'s server-aided-verification queries.

1. **Setup. Signature Queries**. \(B\) responds to these queries in the same way as was described in the case Succ\{E1\}.
2. **Server-Aided-Verification Queries**. At any time when the adversary \(A\) can make a Server-Aided-Verification query of the input \((m_i, \sigma_i)\), \(B\) answers the query in the same way as is described in the first scenario (3\{a\}) of Server-Aided-Verification Queries of the first event E1.

If \(B\) does not abort during the simulation, \(A\) will output a message/signature pair \((m^*, \sigma^*)\) where \(\sigma^* = (\sigma_1^*, \sigma_2^*)\) with the restriction described in Section 3.1. If \(A\) can prove \(\sigma^*\) as a valid signature by the SA-Verify protocol described in Fig. 2, then \(A\) will return \(D^*\) to \(B\) such that \(K_1^* = e(g, g)^{\text{abc}} e(\sigma_2^*, V_0 \prod_{i \in M} V_i)^c\). \(B\) can calculate \(e(g, g)^{\text{abc}} = K_1^*/ e(\sigma_2^*, V_0 \prod_{i \in M} V_i)^c\).

\(B\) can calculate \(e(g, g)^{\text{abc}}\) if and only if:

(a) \(B\) does not abort during **Signature Queries**; the probability that this event happens is \(\Pr[\bigwedge_{i=1}^{q_i} K(m_i) = 1]\).

(b) \(B\) makes a correct guess of the signature pair, which happens with the probability \(\frac{1}{q_i}\). This also guarantees that before \(B\) terminates the simulation, all server-aided-verification queries from \(A\) can be correctly answered.

(c) The above two events happen and \(\alpha_0 + \sum_{i \in M^*} \alpha_i = k\) in the pair \((m^*, \sigma^*)\).

Therefore in this event, the probability that \(B\) can successfully output \(e(g, g)^{\text{abc}}\) is

\[
\Pr[\text{Succ}\{E2\}] = \Pr\left[\bigwedge_{i=1}^{q_i} K(m_i) = 1 \land \alpha_0 + \sum_{i \in M^*} \alpha_i = k\right] \cdot \varepsilon \\
\geq \frac{1}{8q_i(n+1)} \cdot \varepsilon.
\]

Above all, \(B\) can solve the GBDH problem with the probability

\[
\Pr[\text{Succ}] = \Pr[\text{Succ}\{E1\}] \Pr[E1] + \Pr[\text{Succ}\{E2\}] \Pr[E2] \\
\geq \frac{1}{q_i} \cdot \frac{1}{8q_i(n+1)} \cdot \Pr[E1] + \frac{1}{8q_i(n+1)} \cdot \varepsilon \cdot \Pr[E2] \\
\geq \frac{1}{q_i} \cdot \frac{1}{8q_i(n+1)} \cdot \varepsilon \cdot (\Pr[E1] + \Pr[E2]) \\
= \frac{\varepsilon}{8q_i(n+1)}.
\]

Algorithm \(B\)'s running time is the same as \(A\)'s running time plus the time that it takes to respond to \(q_i\) signature queries, \(q_i\) verification queries and compute \(e(g, g)^{\text{abc}}\) from \(A\)'s output. Assume that each takes time \(c(\varepsilon, k)\). Hence, the total running time is at most \(t + c(\varepsilon, k) + \varepsilon\). This completes the proof of **Theorem 2**. 

4.3. **Existentially unforgeable SAV-BLS**

Our second protocol is based on the BLS signature [20]. The description of our protocol is as follows.

1. **ParamGen**. Let \((G_1, G_7)\) be bilinear groups where \(|G_1| = |G_7| = p\), for some prime number \(p \geq 2^k\), \(k\) be the system security number and \(g\) be the generator of \(G_1\). \(e\) denotes the bilinear map \(G_1 \times G_1 \rightarrow G_7\). There is one cryptographic hash function \(H : \{0, 1\}^* \rightarrow G_1\). \(G_1\) is a hash of the string \((G_1, G_7, p, e, H)\).

2. **KeyGen**. The signer picks a random number \(k \in \mathbb{Z}_p^*\) and keeps it as the secret key. The public key is set as \(pk = g^k\).
The SAV-BLS signature scheme is Fig. 3. We now consider each probability individually. Let the BDH problem. Its goal is to compute $e(g^a, g^b, g^c) \in G_1$ which is a random instance of the BDH problem. Its goal is to compute $e(g^a, g^b, g^c) \in G_1$ which is a random instance of the BDH problem.

Proof. We will prove that if there is a $t(q_s, q_v, q_h, \varepsilon)$ adaptively chosen message adversary such that $A$ wins the game defined in Section 3.1 with probability $\varepsilon$, then there exists another algorithm $B$ which can solve a random instance of the bilinear Diffie–Hellman problem in time $t' = t + c_{G_1, G_T}(q_h + 2q_s + 2q_v + 1)$ with success probability $\frac{\varepsilon}{\sqrt{q_h(q_h+1)}}$. This contradicts the $(t', \frac{\varepsilon}{\sqrt{q_h(q_h+1)}})$-BDH assumption on $(G_1, G_T)$.

We employ a technique similar to that in [20] and regard the hash functions $H$ as the random oracle. In the game, $A$ can adaptively make $H$ queries, Signature Queries and Server-Aided Verification Queries. To make the proof clearer, we introduce the notion of a “special pair”. Let $M_v$ be the set of $A$’s server-aided verification queries in the game. A message-signature pair $(m, \sigma) \in M_v$ is a special pair if:

1. by running $\text{SA-Verify}$ $A$ can convince the challenger that $(m, \sigma)$ is a valid message–signature pair; and
2. $m$ has not appeared as one of the signature queries when $A$ makes the server-aided verification query $(m, \sigma)$.

We now define the following two events:

- E1. There is a special pair in $M_v$.
- E2. There is no special pair in $M_v$.

It is clear that either E1 or E2 happens in the game, and thus $\Pr[E1] + \Pr[E2] = 1$.

Let $(G_1, G_T)$ be bilinear groups of prime order $p$. Algorithm $B$ is given $g, g^a, g^b, g^c \in G_1$ which is a random instance of the BDH problem. Its goal is to compute $e(g^a, g^b)$. Algorithm $B$ will simulate the challenger and interact with the adversary $A$ as described below. Let Succ be the event that $B$ solves the given instance of the BDH problem; then we have

$$\Pr[\text{Succ}] = \Pr[\text{Succ} \land E1] + \Pr[\text{Succ} \land E2] = \Pr[\text{Succ}\mid E1] \Pr[E1] + \Pr[\text{Succ}\mid E2] \Pr[E2].$$

We now consider each probability individually.

<table>
<thead>
<tr>
<th>Verifier ($\text{VString: } (r, R)$)</th>
<th>Server ($\text{param}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong></td>
<td></td>
</tr>
<tr>
<td>$(m, \sigma), pk, \text{param}$</td>
<td>$\sigma, R$</td>
</tr>
<tr>
<td>Compute:</td>
<td>$K_1 \leftarrow \epsilon(\sigma, R)$</td>
</tr>
<tr>
<td>Compute:</td>
<td>$K_2 = e(H(m), pk)$</td>
</tr>
<tr>
<td>Output:</td>
<td></td>
</tr>
<tr>
<td>Valid, if $K_1 = K_2$</td>
<td></td>
</tr>
<tr>
<td>Invalid, otherwise.</td>
<td></td>
</tr>
</tbody>
</table>

Note that $R$ is precomputed and the verifier sends the same $R$ to the server in server-aided verification of different message-signature pairs.

![SA-Verify in SAV-BLS with EUF.](image-url)

**Table 2.** Comparisons II.

<table>
<thead>
<tr>
<th>Verification</th>
<th>Pairing</th>
<th>Exponentiation on $G_T$</th>
<th>Map-to-point</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLS</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>SAV-BLS</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

3. **Sign.** For a message $m$ to be signed, the signer uses its secret key to generate the signature $\sigma = H(m)^x$.

4. **Verify.** For a message/signature pair $(m, \sigma)$, everyone can check whether $e(\sigma, g) = e(H(m), pk)$. If the equation holds, output Valid. Otherwise, output Invalid.

5. **SA-Verify-Setup.** Given the system parameters $(G_1, G_T, k, g, p, e, H)$, the verifier $V$ randomly chooses $r \in \mathbb{Z}_p$ and sets $R = g^r$. The VString is $(r, R)$.

6. **SA-Verify.** The verifier $V$ and the server $S$ interact with each other using the protocol described in Fig. 3.

**Computation-Saving.** From Table 2, the verifier in SAV-BLS described above needs to compute one pairing, one exponentiation on $G_T$, and one map-to-point hash. It is obvious that $\Phi - \text{SA-Verify} < \Phi - \text{Verify}$.

**Security Proof of Existentially Unforgeable SAV-BLS**

**Theorem 3.** The SAV-BLS signature scheme is $(t, q_s, q_v, q_h, \varepsilon)$-existentially unforgeable against adaptive chosen message attacks if the $(t + c_{G_1, G_T}(q_h + 2q_s + 2q_v + 1), \frac{\varepsilon}{\sqrt{q_h(q_h+1)}})$-BDH assumption holds on $(G_1, G_T)$. Here, $c_{G_1, G_T}$ is a constant that depends on $(G_1, G_T)$ and $\varepsilon$ is the base of the natural logarithm.

**Proof.** We will prove that if there is a $(t, q_s, q_v, q_h)$ adaptively chosen message adversary such that $A$ wins the game defined in Section 3.1 with probability $\varepsilon$, then there exists another algorithm $B$ which can solve a random instance of the bilinear Diffie–Hellman problem in time $t' = t + c_{G_1, G_T}(q_h + 2q_s + 2q_v + 1)$ with success probability $\frac{\varepsilon}{\sqrt{q_h(q_h+1)}}$. This contradicts the $(t', \frac{\varepsilon}{\sqrt{q_h(q_h+1)}})$-BDH assumption on $(G_1, G_T)$.

We employ a technique similar to that in [20] and regard the hash functions $H$ as the random oracle. In the game, $A$ can adaptively make $H$ queries, Signature Queries and Server-Aided Verification Queries. To make the proof clearer, we introduce the notion of a “special pair”. Let $M_v$ be the set of $A$’s server-aided verification queries in the game. A message-signature pair $(m, \sigma) \in M_v$ is a special pair if:

1. by running $\text{SA-Verify}$ $A$ can convince the challenger that $(m, \sigma)$ is a valid message–signature pair; and
2. $m$ has not appeared as one of the signature queries when $A$ makes the server-aided verification query $(m, \sigma)$.

We now define the following two events:

- E1. There is a special pair in $M_v$.
- E2. There is no special pair in $M_v$.

It is clear that either E1 or E2 happens in the game, and thus $\Pr[E1] + \Pr[E2] = 1$.

Let $(G_1, G_T)$ be bilinear groups of prime order $p$. Algorithm $B$ is given $g, g^a, g^b, g^c \in G_1$ which is a random instance of the BDH problem. Its goal is to compute $e(g^a, g^b)$. Algorithm $B$ will simulate the challenger and interact with the adversary $A$ as described below. Let Succ be the event that $B$ solves the given instance of the BDH problem; then we have

$$\Pr[\text{Succ}] = \Pr[\text{Succ} \land E1] + \Pr[\text{Succ} \land E2] = \Pr[\text{Succ}\mid E1] \Pr[E1] + \Pr[\text{Succ}\mid E2] \Pr[E2].$$

We now consider each probability individually.
The Event Succ|E1. If the event E1 happens, then there is a special pair in \( \mathcal{A} \)'s server-aided verification queries. At the beginning of the simulation, \( \mathcal{B} \) picks a random integer \( j \) in \( \{1, 2, \ldots, q_i\} \), and lets \( j \) be its guess of the index of the first special pair.

1. **Setup.** \( \mathcal{B} \) starts by setting \( pk = g^a \) and \( R = g^c \), where \( g^a, g^c \) are the inputs of the BDH problem, and returns \((G_1, G_2, k, g, p, e, pk) \) to \( \mathcal{A} \).

2. **H queries.** At any time the adversary \( \mathcal{A} \) can request the hash function \( H \) of the input \( m_i \). To respond to these queries, algorithm \( \mathcal{B} \) will maintain an H-List which consists of tuple \((m_i, H(m_i), \alpha_i, coin_i) \) as explained later. For a query \( m_i \), \( \mathcal{B} \) responds as follows.
   (a) If there exists a tuple \((m_i, H(m_i), \alpha_i, coin_i) \) on the H-List, \( \mathcal{B} \) will return \( H(m_i) \) to \( \mathcal{A} \).
   (b) Otherwise, \( \mathcal{B} \) will generate a random coin \( coin_i \in \{0, 1\} \) such that \( Pr[coin_i = 1] = \frac{1}{q_i + 1} \).
   (c) If \( coin_i = 1 \), \( \mathcal{B} \) chooses \( \alpha_i \in \mathbb{Z}_p \) and computes \( H(m_i) = g^b \cdot g^{\alpha_i} \) where \( g^b \) is the input of the BDH problem.
   (d) Otherwise, \( coin_i = 0 \) and \( \mathcal{B} \) chooses \( \alpha_i \in \mathbb{Z}_p \) and computes \( H(m_i) = g^{\alpha_i} \).
   (e) Finally \( \mathcal{B} \) returns \( H(m_i) \) to \( \mathcal{A} \) and adds \((m_i, H(m_i), \alpha_i, coin_i) \) to the list H-List.

3. **Signature Queries.** At any time the adversary \( \mathcal{A} \) can request the signature of the message \( m_i \). We assume that \( m_i \) already appears on the H-List in a tuple \((m_i, H(m_i), \alpha_i, coin_i) \). Otherwise, \( \mathcal{B} \) makes an H query itself to ensure that such a tuple exists on the H-List.
   (a) If \( coin_i = 0 \), then \( \mathcal{B} \) responds to \( \mathcal{A} \) with the signature \( \sigma_i = pk^{\alpha_i} \). Note that \( \sigma_i \) is a valid signature as \( e(\sigma_i, g) = e(pk^{\alpha_i}, g) = e(pk, H(m_i)) \).
   (b) Otherwise, \( \mathcal{B} \) terminates the simulation and reports failure.

4. **Server-Aided Verification Queries.** At any time the adversary \( \mathcal{A} \) can make the server-aided verification of a message–signature pair \((m_i, \sigma_i) \). We assume that \( m_i \) already appears on the H-List. Otherwise, \( \mathcal{B} \) makes an H query itself to ensure such tuple exists on the H-List. Recall that \( \mathcal{B} \) has made a guess at the beginning of the game that the first special pair is the \( j \)th server-aided verification query.
   (a) For the \( i \)th query \((m_i, \sigma_i) \) where \( i < j \):
      - If \( m_i \) has never appeared as one of the signature queries before this server-aided verification query, then \( \mathcal{B} \) will execute the server-aided verification protocol with \( \mathcal{A} \) but output \textit{Invalid} at the end of the protocol, no matter what \( \mathcal{A} \)'s response is in the protocol. It is clear that if \( \mathcal{B} \)'s guess of the first special pair is correct, then \( \mathcal{B} \)'s output will be correct as well.
      - Otherwise, \( \mathcal{A} \) has issued \( m_i \) as one of signature queries and \( \mathcal{B} \) answered with a valid signature which is denoted as \( \sigma'_i \). In this case, \( \mathcal{B} \) will execute the protocol with \( \mathcal{A} \). First, \( \mathcal{B} \) sends \((\sigma_i, R) \) to \( \mathcal{A} \). Then, \( \mathcal{A} \) responds with \( K_i \). Finally, \( \mathcal{B} \) will output \textit{Valid} if \( K_i = e(\sigma'_i, R) \). Otherwise, it outputs \textit{Invalid}.
   (b) Otherwise, \( i = j \) and \((m_j, \sigma_j) \) is a special pair. Let the corresponding tuple on the H-List be \((m_j, H(m_j), \alpha_j, coin_j) \). If \( coin_j = 0 \), \( \mathcal{B} \) reports failure and aborts. Otherwise, \( coin_j = 1 \) and \( H(m_j) = g^b \cdot g^{\alpha_i} \). \( \mathcal{B} \) executes the server-aided verification protocol with \( \mathcal{A} \), by sending \((\sigma_j, R) \) to \( \mathcal{A} \). As the response, \( \mathcal{A} \) will send \( K_j \) to \( \mathcal{B} \). As \((m_j, \sigma_j) \) is a special pair, the server-aided verification protocol \textit{SA-Verify} will output \textit{Valid}. We have \( K_j = e(g, g)^{\alpha \cdot e(\sigma_i, R)} \cdot e(g^a, g^c)^{\alpha} \) as \( pk = g^a, H(m_j) = g^b \cdot g^{\alpha_i} \) and \( R = g^c \). \( \mathcal{B} \) terminates the simulation and outputs \( K_j \cdot e(g^a, g^c)^{\alpha} \) as the solution to the given instance of the BDH problem.

We now compute the probability that \( \mathcal{B} \) solves the BDH problem if \( E1 \) happens. All the following events are required for \( \mathcal{B} \)'s success.

1. \( \mathcal{B} \) does not abort as the result of \( \mathcal{A} \)'s signature queries. This happens with probability \( (1 - 1/(q_i + 1))^q_i \geq 1/\epsilon \). Here, \( \epsilon \) is the base of the natural logarithm.
2. \( \mathcal{B} \) makes a correct guess of the special pair, which happens with probability \( 1/q_i \). This also guarantees that before \( \mathcal{B} \) terminates the simulation, all server-aided verification queries from \( \mathcal{A} \) can be correctly answered.
3. The above two events happen and \( coin_j = 1 \) for the special pair \((m_j, \sigma_j) \). This happens with probability at least \( 1/(q_i + 1) \).

Therefore, if the event \( E1 \) happens, the probability that \( \mathcal{B} \) can solve the random instance of the BDH problem is
\[
Pr[\text{Succ|E1}] \geq \frac{1}{e^{q_i/(q_i + 1)}}.
\]

The Event Succ|E2. If the event \( E2 \) happens, then there is no special pair in \( \mathcal{A} \)'s server-aided verification queries. \( \mathcal{B} \) responds to \( \mathcal{A} \)'s queries as follows.

1. **Setup, H queries, Signature Queries.** \( \mathcal{B} \) responds to these queries in the same way as was described in the case Succ|E1.
2. **Server-Aided Verification Queries.** At any time the adversary \( \mathcal{A} \) can make a server-aided verification query of \((m_i, \alpha_i) \).
   (a) If \( m_i \) has never appeared as one of signature queries before this query, then \( \mathcal{B} \) will execute the server-aided verification protocol with \( \mathcal{A} \) but output \textit{Invalid} at the end of the protocol, no matter what \( \mathcal{A} \)'s response is in the protocol. It is clear that if \( E2 \) happens, then \( \mathcal{B} \)'s output will be correct as well.
   (b) Otherwise, \( \mathcal{A} \) has issued \( m_i \) as one of signature queries and \( \mathcal{B} \) answered with a valid signature which is denoted as \( \sigma'_i \). In this case, \( \mathcal{B} \) will execute the protocol with \( \mathcal{A} \). First, \( \mathcal{B} \) sends \((\sigma_i, R) \) to \( \mathcal{A} \). Then, \( \mathcal{A} \) responds with \( K_i \). Finally, \( \mathcal{B} \) will output \textit{Valid} if \( K_i = e(\sigma'_i, R) \). Otherwise, it will output \textit{Invalid}.
If \( B \) does not abort the simulation, \( A \) will output a message–signature pair \((m^*, \sigma^*)\) with the restriction described in Section 3.1 and convince \( B \) that \( \sigma^* \) is valid by SA-Verify. Let the corresponding tuple on the \( H-\text{List} \) be \((m^*, H(m^*), \sigma^*, \text{coin}^*)\). If \( \text{coin}^* = 0 \), \( B \) reports failure and aborts. Otherwise, \( \text{coin}^* = 1 \) and \( H(m^*) = g^b \cdot g^\sigma^* \). \( B \) then executes the server-aided verification protocol \textbf{SA-Verify} with \( A \). Let \( K_1^* \) be the response of \( A \) in \textbf{SA-Verify}. If \( A \) can successfully prove that \( \sigma^* \) is a valid signature, then \( K_1^* = e(g, g)^{\text{abc}} \cdot e(g^e, g^\sigma^*) \) as \( pk = g^e, H(m^*) = g^b \cdot g^\sigma^* \) and \( R = g^e \). \( B \) thus can output \( K_1^* \cdot e(g^a, g^e)^{-\sigma^*} \) as the solution to the given instance of the BDH problem.

We now compute the probability that \( B \) solves the BDH problem if \( E_2 \) happens. All the following events are required for \( B \)'s success.

1. \( B \) does not abort as the result of \( A \)'s signature queries. This happens with probability \( (1 - 1/(q_s + 1))^{q_b} \geq 1/\epsilon \). Here, \( \epsilon \) is the base of the natural logarithm.
2. \( B \) correctly answers all server-aided verification queries from \( A \). This happens with probability \( 1 \) if \( E_2 \) happens.
3. \( A \) can successfully prove the validity of \( \sigma^* \) by \textbf{SA-Verify}. This happens with the probability \( \epsilon \) if \( B \)'s simulation does not fail.
4. The above three events happen and \( \text{coin}^* = 1 \) for the pair \((m^*, \sigma^*)\). This happens with probability at least \( 1/(q_s + 1) \).

Therefore, if the event \( E_2 \) happens, the probability that \( B \) can solve the given instance of BDH problem is

\[
\Pr[\text{Succ}|E_2] \geq \frac{\epsilon}{\epsilon(q_s + 1)}.
\]

Recall that \( \Pr[\text{Succ}|E_1] \geq \frac{1}{eq_s(q_s+1)} \) and \( \Pr[E_1] + \Pr[E_2] = 1 \); then the probability that \( B \) can solve the given instance of the BDH problem is

\[
\Pr[\text{Succ}] = \Pr[\text{Succ}|E_1] \Pr[E_1] + \Pr[\text{Succ}|E_2] \Pr[E_2] \\
= \frac{1}{eq_s(q_s + 1)} \Pr[E_1] + \frac{\epsilon}{\epsilon(q_s + 1)} \Pr[E_2] \\
\geq \frac{1}{eq_s(q_s + 1)} (\Pr[E_1] + \Pr[E_2]) \\
= \frac{\epsilon}{eq_s(q_s + 1)}.
\]

Algorithm \( B \)'s running time is the same as \( A \)'s running time plus the time it takes to respond to \((q_h + q_c + q_v) \) random oracle queries, \( q_s \) signature queries and \( q_v \) verification queries, and compute \( e(g, g)^{\text{abc}} \) from \( A \)'s output. Each requires at most one pairing operation and one exponentiation which we assume takes time \( c_{(G_1,G_2)} \). Hence, the total running time is at most \( t + c_{(G_1,G_2)} (q_h + 2q_s + 2q_v + 1) \). This completes the proof of Theorem 3. \( \square \)

5. \( \textbf{SAV-} \Sigma \) secure against collusion and adaptive chosen message attacks

As defined earlier (including Definition 3), the server does not know any valid signature of the message \( m \), when it tries to use \textbf{SA-Verify} to convince the verifier that an invalid signature of \( m \) is valid. In this section, we investigate the security of \( \textbf{SAV-} \Sigma \) against collusion between the server and the signer, and propose server-aided verification protocols secure against this attack.

5.1. Security of \( \textbf{SAV-} \Sigma \) against collusion and adaptive chosen message attacks

If we allow the server and the signer to collude, the server will have valid signatures of any messages. Thus, it is impossible to give a unified security notion to capture both EUF-\( \Sigma \) and Soundness-\textbf{SA-Verify} simultaneously. With this in mind, we now define the soundness of the server-aided verification protocol \textbf{SA-Verify} against collusion and adaptive chosen message attacks. In the game, the adversary is given the secret key of the signer.

**Setup.** The challenger \( C \) runs the algorithms \textbf{ParamGen}, \textbf{KeyGen} and \textbf{SA-Verifier-Setup} to obtain system parameter \( \text{param} \), one key pair \((sk, pk)\) and \( V\text{String} \). The adversary \( A \) is given \( \text{param} \) and \((sk, pk)\).

**Queries.** The adversary \( A \) only needs to make \textbf{Server-Aided Verification Queries}. Proceeding adaptively, the adversary \( A \) can make at most \( q_v \) such queries. The challenger \( C \) responds to each query in the same way as was described in Definition 3.

**Output.** The adversary \( A \) will output a message \( m^* \). We denote as \( \Omega_{m^*} \) the set of valid signatures of \( m^* \). The challenger \( C \) chooses a random element \( \sigma^* \) in \( \Omega \setminus \Omega_{m^*} \). That is, \( \sigma^* \) is a random invalid signature of \( m^* \). We say that \( A \) wins the game if

\[
\textbf{SA-Verify}(A, (m^*, \sigma^*, pk, V\text{String})) = \text{Valid}.
\]

Note that the challenge signature is chosen by the challenger, and is not given to the adversary. This is different from the collusion and adaptive chosen message attacks defined in [21].
Table 3
Comparisons III.

<table>
<thead>
<tr>
<th>Verification</th>
<th>Pairing</th>
<th>Exponentiation</th>
<th>Multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waters</td>
<td>2</td>
<td>0</td>
<td>( n(G_1) )</td>
</tr>
<tr>
<td>SAV-Waters</td>
<td>0</td>
<td>Fixed-base:</td>
<td>( 1(G_1) + 1(G_T) )</td>
</tr>
</tbody>
</table>

We define Soundness-\( \text{SA-Verify} \)-Adv \( A \) to be the probability that the adversary \( A \) wins in the above game, taken over the coin tosses made by \( A \) and the challenger.

**Definition 4.** An adversary \( A \) is said to \((t, q_v, \varepsilon)\)-break the soundness of \( \text{SA-Verify} \) in a SAV-\( \Sigma \) if \( A \) runs in time at most \( t \), makes at most \( q_v \) server-aided verification queries, and Soundness-\( \text{SA-Verify} \)-Adv \( A \) is at least \( \varepsilon \). The \( \text{SA-Verify} \) in a SAV-\( \Sigma \) is \((t, q_v, \varepsilon)\)-sound against collision and adaptive chosen message attacks if there exists no adversary that \((t, q_v, \varepsilon)\)-breaks it.

**Definition 5.** SAV-\( \Sigma \) is \((t, q_v, q_s, \varepsilon)\)-secure against collusion and adaptive chosen message attacks if \( \Sigma \) is \((t, q_v, \varepsilon)\)-existentially unforgeable against adaptive chosen message attacks and its server-aided verification protocol \( \text{SA-Verify} \) is \((t, q_v, \varepsilon)\)-sound against collision and adaptive chosen message attacks.

### 5.2. SAV-Waters secure against collusion and adaptive chosen message attacks

In this section, we provide another server-aided verification protocol, which is based on the Waters signature [19] and secure against collusion and adaptive chosen message attacks.

1. **ParamGen, KeyGen, Sign, Verify.** These algorithms are the same as those defined in Section 4.2.
2. **SA-Verifier-Setup.** Given the system parameters \((G_1, G_T, k, g, p, e)\), \( V \) computes \( K_1 = e(g, g) \). Then set \( D = g^d \). The VString is \( K_1 \).
3. **SA-Verify.** The verifier \( V \) and the server \( S \) interact with each other using the protocol described in Fig. 4.

### Computation-Saving
From Table 3, the verifier in SAV-Waters described above only needs to compute one multiplication on \( G_1 \), one (fixed-base) exponentiation on \( G_1 \), two multiplications on \( G_T \) and one (fixed-base) exponentiation on \( G_T \). In particular, there is no pairing operation. Thus, \( \Phi - \text{SA-Verify} < \Phi - \text{Verify} \).

**Security Proof of SAV-Waters Against Collusion and Chosen Message Attacks**

We will show that the server-aided-verification protocol is sound against collusion and adaptive chosen message attacks.

**Theorem 4.** The server-aided-verification protocol described in Fig. 4 is \((t, q_v, \frac{1}{p^2})\)-sound against collusion and adaptive chosen message attacks.

**Proof.** We will prove that the adversary’s probability of proving an invalid signature as valid is \( \frac{1}{p^2} \).

1. **Setup.** The challenger starts by choosing the secret key \( sk = x \in_R \mathbb{Z}_p \), and sets the public key as \( pk = (\bar{v}, PK) \) where \( \bar{v} = (V_i) \) where \( i = 0, 1, \ldots, n \), and \( V_i \in R G_1 \). The challenger also computes \( K_1 = e(g, g) \), then returns \((G_1, G_T, e, p, sk, pk)\) to \( A \).
2. **Server-Aided-Verification Queries.** At any time the adversary \( A \) can make the server-aided-verification query \((m_i, \sigma_i)\).

The challenger executes the protocol \textbf{SA-Verify} with \( A \) as described in Fig. 4.

3. **Output.** After all queries, \( A \) will output a message \( m^* \). Let \( \Omega_{m^*} \) be the valid signature of \( m^* \). As response, the challenger will choose a random element \( \sigma^* = (\sigma_1^*, \sigma_2^*) \in G_1 \setminus \Omega_{m^*} \), that is, \( \sigma^* \) is a random element in the invalid signature space of \( m^* \).

The challenger then executes \textbf{SA-Verify} with \( A \) as described in Fig. 4. The challenger selects \( d^* \in \mathbb{Z}_p \) and computes 
\[
\sigma_1^* = \sigma_1^* \cdot g^{d^*}.
\]
After that, \( \sigma_1^* \) and \( \sigma_2^* \) are sent to \( A \), who will return \( K_2^* \) and \( K_3^* \) as the response. We now show that \( K_2^* = PK \cdot K_2^* \cdot K_1^{d^*} \) happens with probability \( \frac{1}{p-1} \). The following analysis has a similar idea to the proof of the Cramer–Shoup encryption scheme [23].

(a) The element \( \sigma_1^* \) sent to \( A \) does not constrain \((\sigma_1^*, d^*)\). This is because given \( \sigma_1^* \), there are \( (p - 1) \) pairs \((\sigma_1^i, d_i)\) that satisfy the equation \( \sigma_1^i = \sigma_1^* \cdot g^{d_i} \), and \( (\sigma_1^*, d^*) \) chosen by the challenger is just a random one among these \( p - 1 \) pairs. In other words, from the adversary's viewpoint, each \((\sigma_1^i, d_i)\) has equal probability of being \((\sigma_1^*, d^*)\). To make our analysis clearer, we rewrite the equation \( \sigma_1^i = \sigma_1^* \cdot g^{d^*} \) as
\[
DLg\sigma_1^i = DLg\sigma_1^* + d^*.
\]

(b) Suppose \( A \) returns \( K_2^* \) and \( K_3^* \) such that \( K_2^* = PK \cdot K_2^* \cdot K_1^{d^*} \). We rewrite this equation as
\[
DLg\sigma_1^* = DLg\sigma_1^* + d^*.
\]

If Eqs. (1) and (2) are linearly independent, then \( (\sigma_1^*, d^*) \) will be uniquely determined, which happens with probability \( \frac{1}{p-1} \) as \( (\sigma_1^*, d^*) \) is randomly chosen from \( p - 1 \) pairs from the viewpoint of the adversary.

Otherwise, Eqs. (1) and (2) are linearly dependent. This requires that \( DLg\sigma_1^* = DLg\sigma_1^* + d^* \), that is, \( e(\sigma_1^*, g) = PK \cdot K_2^* \). This means that \( \sigma_1^* \) is uniquely determined by \( K_2^* \). As \( \sigma_1^* \) sent to \( A \) does not constrain \( \sigma_1^* \), this happens also with probability \( \frac{1}{p-1} \).

Therefore, the probability that \textbf{SA-Verify} will output \textbf{Valid} is \( \frac{1}{p-1} \). This completes the proof of this theorem. \( \square \)

5.3. **SAV-BLS secure against collusion and adaptive chosen message attacks**

We now give a server-aided verification protocol for the BLS signature [20], which is secure against the collusion and adaptive chosen message attacks.

1. **ParamGen, KeyGen, Sign, Verify.** These algorithms are the same as those defined in Section 4.3.

2. **SA-Verifier-Setup.** Given the system parameters \((G_1, G_T, k, g, p, e, H)\), the verifier \( V \) computes \( K_1 = e(g, g) \). The VString is \( K_1 \).

3. **SA-Verify.** The verifier \( V \) and the server \( S \) interact with each other using the protocol described in Fig. 5.

**Computation-Saving.** From Table 4, the verifier in SAV-BLS described above only needs to compute one multiplication on \( G_1 \), one (fixed-base) exponentiation on \( G_1 \), one multiplication on \( G_T \) and one (fixed-base) exponentiation on \( G_T \). In particular, there is no pairing or map-to-point operation. Thus, \( \Phi$-\textbf{SA-Verify} < \Phi$-\textbf{Verify}.

**Security Proof of SAV-BLS Against Collusion and Chosen Message Attacks.** We only need to show that the server-aided verification protocol in Fig. 5 is sound against collusion and adaptive chosen message attacks.

**Theorem 5.** The server-aided verification protocol described in Fig. 5 is \((t, q_v, \frac{1}{p-1})\)-sound against collusion and adaptive chosen message attacks.

**Proof.** We prove that the adversary's probability of proving an invalid signature as valid is \( \frac{1}{p-1} \), without any complexity assumption.

1. **Setup.** The challenger starts by choosing the secret key \( sk = x \in \mathbb{Z}_p \), sets the public key as \( pk = g^x \) and computes \( K_1 = e(g, g) \), returns \( (G_1, G_T, k, g, p, e, H) \) and \( (sk, pk) \) to \( A \).

2. **Server-Aided Verification Queries.** At any time the adversary \( A \) can make the server-aided verification query \((m_i, \sigma_i)\).

The challenger executes the protocol \textbf{SA-Verify} with \( A \) as described in Fig. 5.

3. **Output.** After all the queries, \( A \) will output a message \( m^* \). As a response, the challenger will choose a random element \( \sigma^* \in G_1 \setminus \{H(m^*)\} \), that is, \( \sigma^* \) is a random element in the invalid signature space of \( m^* \).

The challenger then executes \textbf{SA-Verify} with \( A \) as described in Fig. 5. The challenger selects \( r^* \in \mathbb{Z}_p \) and computes \( \sigma^* = \sigma^* \cdot g^{r^*} \). After that, \( \sigma^* \) is sent to \( A \), who will return \( K_2^* \) and \( K_3^* \) as the response. We now show that \( K_2^* = K_2^* \cdot K_1^{d^*} \) happens with probability \( \frac{1}{p-1} \). The following analysis use a technique similar to that in the proof of the Cramer–Shoup encryption scheme [23].

(a) The element \( \sigma^* \) sent to \( A \) does not constrain the distribution of \((\sigma^*, r^*)\). This is because given \( \sigma^* \), there are \( (p - 1) \) pairs \((\sigma_i, r_i)\) that satisfy the equation \( \sigma^* = \sigma_i \cdot g^{r_i} \), and \((\sigma^*, r^*) \) chosen by the challenger is just a random one among these \( p - 1 \) pairs. In other words, from the adversary's viewpoint, each \((\sigma_i, r_i)\) has equal probability of being \((\sigma^*, r^*)\). To make our analysis clearer, we rewrite the equation \( \sigma^* = \sigma^* \cdot g^{r^*} \) as
\[
DLg\sigma^* = DLg\sigma^* + r^*.
\]
Suppose $A$ returns $K^*_{2}$ and $K^*_{3}$ such that $K^*_{2} = K^*_{3} \cdot K^*_1$. We rewrite this equation as

\[ DL_{e}(g, g) K^*_{2} = DL_{e}(g, g) K^*_{3} + r^*. \]

(b) If $DL_g \sigma^* \neq DL_{e}(g, g) K^*_3$, then Eqs. (1) and (2) are linearly independent. It follows that $r^*$ satisfies Eq. (2) with probability $\frac{1}{p-1}$, as $(\sigma^*, r^*)$ is randomly chosen from $p - 1$ pairs from the viewpoint of the adversary.

Otherwise, $DL_g \sigma^* = DL_{e}(g, g) K^*_3$, that is, $e(\sigma^*, g) = K^*_3$. This means that $\sigma^*$ is uniquely determined by $K^*_3$. As $\sigma^*$ sent to $A$ does not constrain the distribution of $\sigma^*$, this happens also with probability $\frac{1}{p-1}$.

Therefore, the probability that $\text{SA-Verify}$ will output $\text{Valid}$ is $\frac{1}{p-1}$. This completes the proof of this theorem. \hfill $\square$

**Remark.** Recently a new type of “collusion attacker” was defined in [24]. In the new definition, an attacker is given a key pair $(sk_f, pk_f)$ and a public key $pk$ while the challenger keeps the corresponding private key $sk$ as secret. The attacker is said to break the soundness of the SAV protocol if he/she can find a message/signature pair $(m^*, \sigma^*)$ which is valid under the public key $pk_f$ and can be proved as valid under $pk$ via the SAV protocol. As one can see, the adversary defined in [24] actually belongs to those defined in Section 3.1, i.e., the adversary does not have the private key but can choose the challenge message/signature pair. (Also notice that the key pair $(sk_f, pk_f)$ can also be generated by the adversaries defined in Section 5.1.) This is different from the collusion attacks defined in Section 5.1, where the adversary is given the private $sk$ but is not allowed to choose the challenge signature $\sigma^*$. As shown in [24], SAV protocols secure against collusion attacks defined in this section might be existentially forgeable, since adversaries in these two notions are different. It is certainly more desirable if SAV protocols are secure against collusion attacks where adversaries are also allowed to choose the challenge message/signature pair. These protocols will be investigated in the following section.

### 5.4 SAV-Σ secure against strong collusion and adaptive chosen message attacks

As we can see in Section 5.1, it is the challenger who chooses an invalid signature of the message $m^*$ (where $m^*$ is chosen by the adversary). This section considers strong collusion and adaptive chosen message attacks, where the adversary has the ability to choose the invalid signature under the message $m^*$. The concrete game is defined as follows.

**Setup.** The challenger $C$ runs the algorithms $\text{ParamGen}$, $\text{KeyGen}$ and $\text{SA-Verifier-Setup}$ to obtain system parameter $\text{param}$, one key pair $(sk, pk)$ and $\text{VString}$. The adversary $A$ is given $\text{param}$ and $(sk, pk)$.
Fig. 6. SA-Verify in SAV-BLS with soundness.

\[ \text{Verifier (VString: } K_1) \quad \text{Server (param)} \]

Input:
\((m, \sigma), pk, \text{param} \)

Compute:
\[ r_1, r_2 \in \mathbb{Z}_p, \quad \sigma' = \sigma^{r_1} \cdot g^{r_2} \quad \xrightarrow{m, \sigma', pk} \]
\[ K_2 = e(\sigma', g) \]
\[ K_3 = e(H(m), pk) \]

Output:
Valid, if \( K_2 = K_3^{r_1} \cdot K_1^{r_2} \)
Invalid, otherwise.

Table 5
Comparisons V.

<table>
<thead>
<tr>
<th>Verification</th>
<th>Pairing</th>
<th>Exponentiation</th>
<th>Multiplication</th>
<th>Map-to-point</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLS</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>SAV-BLS</td>
<td>1</td>
<td>2(G_1) + 2(G_T)</td>
<td>1(G_1) + 1(G_T)</td>
<td>1</td>
</tr>
</tbody>
</table>

**Queries.** The adversary \( \mathcal{A} \) only needs to make **Server-Aided Verification Queries.** Proceeding adaptively, the adversary \( \mathcal{A} \) can make at most \( q_v \) such queries. The challenger \( \mathcal{C} \) responds to each query in the same way as was described in Definition 3.

**Output.** The adversary \( \mathcal{A} \) will output a message \( m^* \) and choose a random element \( \sigma^* \in \Omega \setminus \Omega_{m^*} \), where \( \Omega_{m^*} \) denotes the set of valid signatures of \( m^* \). We say that \( \mathcal{A} \) wins the game if \( \text{SA-Verify}(\mathcal{A}, \mathcal{C}(m^*, \sigma^*, pk, \text{VString})) = \text{Valid} \).

We define **Strong-Soundness-**\( \text{SA-Verify} \)-Adv \( \mathcal{A} \) to be the probability that the adversary \( \mathcal{A} \) wins in the above game, taken over the coin tosses made by \( \mathcal{A} \) and the challenger.

**Definition 6.** An adversary \( \mathcal{A} \) is said to \((t, q_v, \varepsilon)\)-strongly break the soundness of **SA-Verify** in a SAV-\( \Sigma \) if \( \mathcal{A} \) runs in time at most \( t \), makes at most \( q_v \) server-aided verification queries, and \( \text{Strong-Soundness-} \text{SA-Verify} - \text{Adv} \mathcal{A} \) is at least \( \varepsilon \). The **SA-Verify** in a SAV-\( \Sigma \) is \((t, q_v, \varepsilon)\)-sound against strong collusion and adaptive chosen message attacks if there exists no adversary that \((t, q_v, \varepsilon)\)-breaks it.

**Definition 7.** SAV-\( \Sigma \) is \((t, q_v, \varepsilon)\)-secure against strong collusion and adaptive chosen message attacks if \( \Sigma \) is \((t, q_v, \varepsilon)\)-existentially unforgeable against adaptive chosen message attacks and its server-aided verification protocol **SA-Verify** is \((t, q_v, \varepsilon)\)-sound against strong collusion and adaptive chosen message attacks.

**SAV-BLS Secure Against Strong Collusion and Adaptive Chosen Message Attacks**
We now give a server-aided verification protocol for BLS signature [20], which is secure against the strong collusion and adaptive chosen message attacks.

1. **ParamGen, KeyGen, Sign, Verify.** These algorithms are the same as those defined in Section 4.3.
2. **SA-Verifier-Setup.** Given the system parameters \((G_1, G_T, k, g, p, e, H)\), the verifier \( V \) computes \( K_1 = e(g, g) \). The VString is \( K_1 \).
3. **SA-Verify.** The verifier \( V \) and the server \( S \) interact with each other using the protocol described in Fig. 5.

**Computation-Saving.** From Table 5, the verifier in SAV-BLS described above needs to compute one multiplication on \( G_1 \), one (fixed-base) exponentiation on \( G_1 \), one exponentiation on \( G_1 \), one multiplication on \( G_T \), one (fixed-base) exponentiation on \( G_T \), one exponentiation on \( G_T \) and one pairing. This is also less than the computational cost required in the original verify algorithm.

**Theorem 6.** The server-aided verification protocol described in Fig. 6 is \((t, q_v, \varepsilon)\)-strong-sound against collusion and adaptive chosen message attacks.
Proof. We prove that the adversary’s probability of proving an invalid signature as valid is $\frac{1}{p}$, without any complexity assumption.

1. Setup. The challenger starts by choosing the secret key $sk = x \in \mathbb{Z}_p$, sets the public key as $pk = g^x$ and computes $K_i = e(g, g)$, then returns $(G_1, G_T, k, g, p, e, H)$ and $(sk, pk)$ to $A$.

2. Server-Aided Verification Queries. At any time the adversary $A$ can make the server-aided verification query $(m, \sigma_i)$. The challenger executes the protocol $\text{SA-Verify}$ with $A$ as described in Fig. 6.

3. Output. After all queries, $A$ will output a message $m^*$ along with a random element $\sigma^* \in G_1 \setminus \{H(m^*)^k\}$, that is, $\sigma^*$ is a random element in the invalid signature space of $m^*$.

The challenger then executes $\text{SA-Verify}$ with $A$ as described in Fig. 6. The challenger selects $r_{i1}^*, r_{i2}^* \in \mathbb{Z}_p$ and computes $\sigma'^* = \sigma_{i1}^* \cdot g^{r_{i1}^*}$. After that, $\sigma'^*$ is sent to $A$, who will return $K_2^*$ as the response. We now show that $K_2^* = K_{3i1}^* \cdot K_{1i}^*$ happens with probability $\frac{1}{p}$. The following analysis uses a technique similar to that in the proof of the Cramer–Shoup encryption scheme [23].

(a) The element $\sigma'^*$ sent to $A$ does not constrain the distribution of $(r_{i1}^*, r_{i2}^*)$. This is because given $\sigma'^*$, there are $p$ pairs $(r_{i1}, r_{i2})$ that satisfy the equation $\sigma'^* = \sigma_{i1}^{r_{i1}} \cdot g^{r_{i2}}$, and $(r_{i1}^*, r_{i2}^*)$ chosen by the challenger is just a random one among these $p$ pairs. In other words, from the adversary’s viewpoint, each $(r_{i1}, r_{i2})$ has equal probability of being $(r_{i1}^*, r_{i2}^*)$. To make our analysis clearer, we rewrite the equation $\sigma'^* = \sigma_{i1}^{r_{i1}} \cdot g^{r_{i2}}$ as

$$DL_{g} \sigma'^* = r_{i1}^* \cdot DL_{g} \sigma_{i1} + r_{i2}^*. \quad (1)$$

(b) Suppose $A$ returns $K_2^*$ such that $K_2^* = K_{3i1}^* \cdot K_{1i}^*$. We rewrite this equation as

$$DL_{e(g, g)} K_2^* = r_{i1}^* \cdot DL_{e(g, g)} K_{3i1}^* + r_{i2}^*. \quad (2)$$

As required, $\sigma^*$ is an invalid signature of $m^*$, i.e. $DL_{g} \sigma^* \neq DL_{e(g, g)} K_2^*$. It follows that Eqs. (1) and (2) are linearly independent and $(r_{i1}^*, r_{i2}^*)$ will be uniquely determined. This happens with probability $\frac{1}{p}$ as $(r_{i1}^*, r_{i2}^*)$ is randomly chosen from $p$ pairs from the viewpoint of the adversary.

Therefore, the probability that $\text{SA-Verify}$ will output $\text{Valid}$ is $\frac{1}{p}$. This completes the proof of this theorem. \qed

6. Conclusion

We formally defined the existential unforgeability of server-aided verification signatures to expand the existing security requirements in server-aided verification signatures. We analyzed the Girault–Lefranc scheme from Asiacrypt 2005 and proposed the first server-aided verification Waters signature (whose existential unforgeability does not rely on the random oracle model) and server-aided verification BLS signature (whose existential unforgeability is proved in the random oracle model). We defined the security of server-aided verification signatures under collusion attacks and strong collusion attacks. Concrete constructions secure against such attacks were also presented.

References


