

## Chapter 2

### Preliminaries

**Abstract** In this chapter we provide formal definitions for verifiable computing schemes and their relevant properties. More precisely, first, we define *verifiable computing schemes* in general and *privately verifiable computing schemes* and *publicly verifiable computing schemes* in particular. Then, we provide a definition for *weak* and *adaptive security*. Following, we discuss the different types of privacy protection, i.e. *input privacy w.r.t. the server*, *input privacy w.r.t. the verifier*, *output privacy w.r.t. the server*, *output privacy w.r.t. the verifier* and give a definition for each property. Finally, we define efficiency distinguishing between *efficiency* and *amortized efficiency*. Many verifiable computing schemes presented in the subsequent chapters are constructed with the help of cryptographic primitives that come with additional definitions for the underlying hardness assumptions. However, since these are very specific to the individual solutions they are presented in Appendix A.

### 2.1 Verifiable Computation

In this work we will always consider the following scenario. A client  $C$  provides some input  $x$  to a server  $S$ . Then  $S$  is asked to evaluate a (where appropriate encoded) function  $f$  on input  $x$ .  $S$  will do the computation and then send the result  $y$  to a verifier  $V$ . To prove the correctness of the result to  $V$ , i.e. to prove that  $y$  is indeed equal to  $f(x)$ , a verifiable computing scheme can be used. In the following we use the definition of a non-interactive verifiable computing scheme introduced by Gennaro et al. [2].

**Definition 2.1 (Verifiable Computing Scheme)** A *Verifiable Computing Scheme*  $VC$  is a tuple of the following probabilistic polynomial-time (PPT) algorithms:

**KeyGen**( $1^\lambda, f$ ) : The probabilistic key generation algorithm takes a security parameter  $\lambda$  and the description of a function  $f$ . It generates a secret key  $sk$ , a corresponding verification key  $vk$ , and a public evaluation key  $ek$  (that encodes the target function  $f$ ) and returns all these keys.

**ProbGen**( $sk, x$ ) : The problem generation algorithm takes a secret key  $sk$  and data  $x$ . It outputs a public value  $\sigma_x$  which encodes the data  $x$  and a corresponding decoding value  $\rho_x$ .

**Compute**( $\mathbf{ek}, \sigma_x$ ) : The computation algorithm takes the evaluation key  $\mathbf{ek}$  and the encoded input  $\sigma_x$ . It outputs an encoded version  $\sigma_y$  of the function's output  $y = f(x)$ .

**Verify**( $\mathbf{vk}, \rho_x, \sigma_y$ ) : The verification algorithm obtains a verification key  $\mathbf{vk}$  and the decoding value  $\rho_x$ . It converts the encoded output  $\sigma_y$  into the output of the function  $y$ . If  $y = f(x)$  holds, it returns  $y$  or outputs  $\perp$  indicating that  $\sigma_y$  does not represent a valid output of  $f$  on  $x$ .

**Definition 2.2 (Correctness)** A verifiable computing scheme  $VC$  is correct if for any choice of  $f$  and output  $(\mathbf{sk}, \mathbf{vk}, \mathbf{ek}) \leftarrow \text{KeyGen}(1^\lambda, f)$  of the key generation algorithm it holds that  $\forall x \in \text{Domain}(f)$ , if  $(\sigma_x, \rho_x) \leftarrow \text{ProbGen}(\mathbf{sk}, x)$  and  $y \leftarrow \text{Compute}(\mathbf{ek}, \sigma_x)$ , then  $y = f(x) \leftarrow \text{Verify}(\mathbf{vk}, \rho_x, \sigma_y)$ .

In the original work on non-interactive verifiable computing Gennaro et al. [2] only considered privately verifiable computing schemes as defined below.

**Definition 2.3 (Privately Verifiable Computing Scheme)** If  $\mathbf{sk} = \mathbf{vk}$  and  $C$  needs to keep  $\rho_x$  private,  $VC$  is called a *privately verifiable computing scheme*.

Clearly, such a scheme requires the client to run the verification algorithm. Later in [3], Parno et al. introduced the notion of publicly verifiable computing schemes.

**Definition 2.4 (Publicly Verifiable Computing Scheme)** If  $\mathbf{sk} \neq \mathbf{vk}$ ,  $VC$  is called a *publicly verifiable computing scheme*.

It allows to hand out  $\mathbf{vk}$  to third parties without revealing  $\mathbf{sk}$ . Therefore, everyone with knowledge of  $\mathbf{vk}$  and  $\rho_x$  can verify the correctness of the server's computation.

Intuitively the difference between the two notions is that in privately verifiable computing the client keeps its verification key secret. It follows that only the client can act as verifier. Note that in privately verifiable computing schemes revealing the verification key often leads to a loss of security. More precisely, knowledge of the verification key allows the server to compute a wrong result leading to a correct verification proof. In publicly verifiable computing, on the other hand, knowledge of the verification key does not help a malicious server to forge an incorrect result. Thus, it can be published allowing not only the client but anyone to act as verifier and to check the correctness of a performed computation.

## 2.2 Properties of Verifiable Computing Schemes

In this section a definition for security, privacy, and efficiency is given. We will mainly follow the approach of Gennaro et al. [2], who were the first to define verifiable computing schemes. In addition, we also integrate some later proposals to obtain stronger security definitions, e.g. adaptive security presented in [1].

### 2.2.1 Security

Intuitively a verifiable computing scheme  $VC$  is secure, if a malicious server cannot persuade the verification algorithm to output  $y^* \neq f(x)$  except with negligible probability. Formally, we define the following two experiments. We distinguish between two types of adversaries, a weak adversary and an adaptive adversary. The weak adversary [2] only has oracle access to **ProbGen** but is not allowed to call **Verify** in the privately verifiable computing setting. It can only try once to have an incorrect result verified as correct but must never learn the client's acceptance bit, since this information might be used to produce subsequent forgeries. An adaptive adversary [1] can run  $\mathbf{EXP}_A^{\text{Verify}}$  multiple times, by calling  $\mathbf{EXP}_{\text{Adapt}}^{\text{Verify}}$ , and learn about the client's acceptance bit and adapt its forgeries accordingly.

Experiment  $\mathbf{EXP}_A^{\text{Verify}}[VC, f, \lambda]$  :

```

(sk, vk, ek)  $\leftarrow$  KeyGen( $1^\lambda, f$ )
for  $i = 1, \dots, \ell = \text{poly}(\lambda)$  do
   $x_i \leftarrow A(\text{ek}, x_1, \dots, x_{i-1}, \sigma_1, \dots, \sigma_{i-1})$ 
   $(\sigma_i, \rho_i) \leftarrow \text{ProbGen}(\text{sk}, x_i)$ 
end for
 $(i, \sigma_y^*) \leftarrow A(\text{ek}, x_1, \dots, x_\ell, \sigma_1, \dots, \sigma_\ell)$ 
 $y^* \leftarrow \text{Verify}(\text{vk}, \rho_i, \sigma_y^*)$ 
if  $y^* \neq \perp \wedge y^* \neq f(x)$  then
  return 1
else
  return 0
end if

```

Experiment  $\mathbf{EXP}_A^{\text{AdaptVerify}}[VC, f, \lambda]$ :

```

(sk, vk, ek)  $\leftarrow$  KeyGen( $f, 1^\lambda$ )
for  $j = 1, \dots, m = \text{poly}(\lambda)$  do
  for  $i = 1, \dots, \ell = \text{poly}(\lambda)$  do
     $x_i \leftarrow A(\text{ek}, x_1, \dots, x_{i-1}, \sigma_1, \dots, \sigma_{i-1}, \delta_1, \dots, \delta_{j-1})$ 
     $(\sigma_i, \rho_i) \leftarrow \text{ProbGen}(\text{sk}, x_i)$ 
  end for
   $(i, \sigma_y^*) \leftarrow A(\text{ek}, x_1, \dots, x_\ell, \sigma_1, \dots, \sigma_\ell, \delta_1, \dots, \delta_{j-1})$ 
   $y^* \leftarrow \text{Verify}(\text{vk}, \rho_i, \sigma_y^*)$ 
  if  $y^* \neq \perp \wedge y^* \neq f(x)$  then
     $\delta_j := 1$ 
  else
     $\delta_j := 0$ 
  end if
end for
if  $\exists j$  such that  $\delta_j = 1$  then
  return 1
else

```

**return 0**  
**end if**

In the non-adaptive case the adversaries  $A$ 's advantage is defined as

$$\text{Adv}_A^{\text{Verify}}(VC, f, \lambda) = \Pr \left[ \mathbf{EXP}_A^{\text{Verify}}[VC, f, \lambda] = 1 \right].$$

So in practice this type of adversary is acceptable if a client aborts the protocol once it detects an incorrect result.

An adaptive adversaries  $A$ 's advantage is defined as

$$\text{Adv}_A^{\text{AdaptVerify}}(VC, f, \lambda) = \Pr \left[ \mathbf{EXP}_A^{\text{AdaptVerify}}[VC, f, \lambda] = 1 \right].$$

From this the security definition for verifiable computing schemes follows.

**Definition 2.5 (Security)** A verifiable computing scheme  $VC$  is (weakly) secure if

$$\text{Adv}_A^{\text{Verify}}(VC, f, \lambda) \leq \text{negl}(\lambda)$$

and adaptively secure if

$$\text{Adv}_A^{\text{AdaptVerify}}(VC, f, \lambda) \leq \text{negl}(\lambda).$$

### 2.2.2 Privacy

Verifiable computing can guarantee the integrity of a computation. Another desirable property is to protect the secrecy of the client's inputs towards the server and when using a publicly verifiable scheme also towards the verifiers. To formally define *input privacy w.r.t the server* we define the following experiment. We use the oracle  $O^{\text{ProbGen}(\text{sk}, x)}$  which calls  $\text{ProbGen}(\text{sk}, x)$  to obtain  $(\sigma_x, \rho_x)$  and only returns the public part  $\sigma_x$ .

Experiment  $\mathbf{EXP}_A^{\text{PrivacyServer}}[VC, f, \lambda]$   
 $(\text{sk}, \text{vk}, \text{ek}) \leftarrow \text{KeyGen}(f, 1^\lambda)$   
 $(x_0, x_1) \leftarrow A^{O^{\text{ProbGen}(\text{sk}, \cdot)}}(\text{ek})$   
 $(\sigma_0, \rho_0) \leftarrow \text{ProbGen}(\text{sk}, x_0)$   
 $(\sigma_1, \rho_1) \leftarrow \text{ProbGen}(\text{sk}, x_1)$   
 $b \xleftarrow{\$} \{0, 1\}$   
 $b^* \leftarrow A^{O^{\text{ProbGen}(\text{sk}, \cdot)}}(\text{ek}, x_0, x_1, \sigma_b)$   
**if**  $b^* = b$  **then**  
  **return 1**  
**else**  
  **return 0**  
**end if**

In this experiment, the adversary first receives the public evaluation key for the scheme. Then, it selects two inputs  $x_0, x_1$  and is given the encoding of one of the two inputs chosen at random. The adversary then must determine which input has been encoded. Note that during this process the adversary is allowed to request the encoding of any input of its choice. We define an adversaries  $A$ 's advantage as

$$\text{Adv}_A^{\text{PrivacyServer}}(VC, f, \lambda) = \left| \Pr \left[ \mathbf{EXP}_A^{\text{PrivacyServer}}[VC, f, \lambda] = 1 \right] - 1/2 \right|.$$

**Definition 2.6 (Input Privacy w.r.t. the Server)** A verifiable computing scheme  $VC$  provides input privacy if

$$\text{Adv}_A^{\text{PrivacyServer}}(VC, f, \lambda) \leq \text{negl}(\lambda).$$

Besides *input privacy* a verifiable computing scheme can also provide privacy with respect to the data output. This so called *output privacy* can be defined by an analogous experiment and is omitted here.

If we have a publicly verifiable computing scheme a third party verifier might try to learn about the input data from the publicly available verification data. To formally define *input privacy w.r.t a third party verifier* we define the following experiment.

Experiment  $\mathbf{EXP}_A^{\text{PrivacyVerifier}}[VC, f, \lambda]$   
 $(\text{sk}, \text{vk}, \text{ek}) \leftarrow \text{KeyGen}(f, 1^\lambda)$   
 $(x_0, x_1) \leftarrow A^{O^{\text{ProbGen}(\text{sk}, \cdot)}}(\text{vk})$   
 $(\sigma_0, \rho_0) \leftarrow \text{ProbGen}(\text{sk}, x_0)$   
 $(\sigma_1, \rho_1) \leftarrow \text{ProbGen}(\text{sk}, x_1)$   
 $b \xleftarrow{\$} \{0, 1\}$   
 $b^* \leftarrow A^{O^{\text{ProbGen}(\text{sk}, \cdot)}}(\text{vk}, x_0, x_1, \sigma_b, \rho_b)$   
**if**  $b^* = b$  **then**  
    **return** 1  
**else**  
    **return** 0  
**end if**

In this experiment, the adversary first receives the public verification key for the scheme. Then, it selects two inputs  $x_0, x_1$  and is given the encoding of one of the two inputs chosen at random. The adversary then must determine which input has been encoded. Note that during this process the adversary is allowed to request the encoding of any input of its choice.

We define an adversaries  $A$ 's advantage as

$$\text{Adv}_A^{\text{PrivacyVerifier}}(VC, f, \lambda) = \left| \Pr \left[ \mathbf{EXP}_A^{\text{PrivacyVerifier}}[VC, f, \lambda] = 1 \right] - 1/2 \right|.$$

**Definition 2.7 (Input Privacy w.r.t. the Verifier)** A verifiable computing scheme  $VC$  provides input privacy if

$$\text{Adv}_A^{\text{Privacy}_{\text{Verifier}}}(VC, f, \lambda) \leq \text{negl}(\lambda).$$

### 2.2.3 Efficiency

Finally we are interested in using verifiable computing schemes by means of delegating computations. For this we want the work performed by the client and the verifier to be less than computing the function on their own.

**Definition 2.8 (Efficiency)** A verifiable computing scheme provides *efficiency* if for any  $x$  and any  $\sigma_y$ , the time required for  $\text{KeyGen}(1^\lambda, f)$  plus the time required for  $\text{ProbGen}(\text{sk}, x)$  plus the time required for  $\text{Verify}(\text{vk}, \rho_x, \sigma_y)$  is  $o(T)$ , where  $T$  is the time required to compute  $f(x)$ .

A slightly relaxed definition is the following.

**Definition 2.9 (Amortized Efficiency)** A verifiable computing scheme provides *amortized efficiency* if it permits efficient verification. This implies that for any  $x$  and any  $\sigma_y$ , the time required for  $\text{Verify}(\text{vk}, \rho_x, \sigma_y)$  is  $o(T)$ , where  $T$  is the time required to compute  $f(x)$ .

Note that in literature amortized efficiency has been defined ambiguously. We use here a broad version that ensures that the minimal requirements for outsourceability are met.

Intuitively the difference between efficiency and amortized efficiency is the cost of the preprocessing phase. Efficient verifiable computing schemes allow a verifier to verify the correctness of a computation more efficiently than performing the computation by itself, including the preprocessing phase performed by the client. Some verifiable computing schemes have an expensive preprocessing phase, but still provide an efficient verification phase. Since the preprocessing phase only has to be performed once and might not be time critical in many applications, we classify them as verifiable computing schemes providing amortized efficiency.

One aspect that also impacts the practicality of all verifiable computing schemes is the server's overhead to evaluate a computation using **Compute** versus natively executing it. Note that this does not affect the computation complexity for the client or the verifier. So far in literature this aspect has not been rigorously covered and is therefore not considered in our efficiency analysis.

## References

1. S. Benabbas, R. Gennaro, Y. Vahlis, Verifiable delegation of computation over large datasets, in *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Proceedings*, Santa Barbara, CA, 14–18 August 2011, pp. 111–131

2. R. Gennaro, C. Gentry, B. Parno, Non-interactive verifiable computing: outsourcing computation to untrusted workers, in *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Proceedings*, Santa Barbara, CA, 15–19 August 2010, pp. 465–482
3. B. Parno, M. Raykova, V. Vaikuntanathan, How to delegate and verify in public: verifiable computation from attribute-based encryption, in *Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Proceedings*, Taormina, Sicily, 19–21 March 2012, pp. 422–439

Privately and Publicly Verifiable Computing Techniques  
A Survey

Demirel, D.; Schabhüser, L.; Buchmann, J.

2017, XII, 64 p., Softcover

ISBN: 978-3-319-53797-9