Verifiable Delay Function

–

Shanghai Jiao Tong University

December 3, 2020

Time-lock puzzle

Time-lock puzzle

The goal of time-lock puzzle is to “encrypt” a message that cannot be decrypted by **anyone**, not even the sender, until a pre-determined amount of time has passed.

Application scenario

Time-lock puzzle

Implementations of time-lock puzzle

There are two natural approaches to implementing:

- Using trusted agents who promise not to reveal the information until the pre-determined time.
- Using computational problems that cannot be solved in a certain amount of time.

Discussion

- The agents must be trustworthy, otherwise the information may leak before the time.
- Users can use parallelism and designated hardware to accelerate the computational problems.

Distributed solutions

Secret sharing

The information can be separated into different agents using secret sharing. By this method, the assumption that agents are trusted can be mitigated.
However, this method needs third parties and may suffers DoS attack.

Computational methods

Computational methods is more reliable as its security relies on mathematics problems. Now there are many kinds of solutions such as memory-hard functions (MHFs), proof of sequential work (PoSW) and **verifiable delay functions (VDFs)**.

Verifiable Delay Function

Definition

simple introduction

A VDF is a function whose evaluation requires running a given number of sequential steps, yet the result can be efficiently verified

A VDF $V = (\textit{Setup}, \textit{Eval}, \textit{Verify})$ is a triple of algorithms as follows:

- *Setup*$(\lambda, t) \rightarrow \mathbf{pp} = (ek, vk)$ is a randomized algorithm where λ is security parameter, t is desired puzzle difficulty and $\mathbf{pp}$ consists of an evaluation key ek and a verification key vk.
- *Eval*$(ek, x) \rightarrow (y, \pi)$ takes an input $x \in \mathcal{X}$ and produces a **unique** output $y \in \mathcal{Y}$ and a (possible empty) proof π.
- *Verify*$(vk, x, y, \pi) \rightarrow \{0, 1\}$ is a deterministic algorithm.

More about the definition

- *Setup* might need randomness, leading to a scheme requiring a trusted setup.
- *Eval* may use random bits to generate π but not to compute y.
- For all valid $\mathbf{pp}$ and all $x \in \mathcal{X}$, *Eval* must run in parallel time t with *poly*$(\log(t), \lambda)$ processors.
- *Verify* must run in total time polynomial in $\log(t)$ and λ. Note that *Verify* is much faster than *Eval*.

Properties

Correctness

A VDF V is correct if for all λ, t, parameters $(ek, vk) \overset{\$}{\leftarrow}$ *Setup*(λ, t), and all $x \in \mathcal{X}$, if $(y, \pi) \leftarrow$ *Eval*(ek, x) then *Verify*$(vk, x, y, \pi) = 1$

Soundness

A VDF is sound if for all algorithm $\mathcal{A}$ that run in time $O(\textit{poly}(t, \lambda))$

$$\Pr\left[\begin{array}{l|l} \textit{Verify}(vk, x, y, \pi) = 1 & \mathbf{pp} = (ek, vk) \overset{\$}{\leftarrow} \textit{Setup}(\lambda, t) \\ y \neq \textit{Eval}(ek, x) & (x, y, \pi) \leftarrow \mathcal{A}(\lambda, \mathbf{pp}, t) \end{array}\right] = \textit{negl}(\lambda)$$

Properties

$$\boxed{\begin{array}{l} \mathbf{pp} \xleftarrow{\$} \textit{Setup}(\lambda, t) \\ L \leftarrow \mathcal{A}_0(\lambda, \mathbf{pp}, t) \\ x \xleftarrow{\$} \mathcal{X} \\ y_{\mathcal{A}} \leftarrow \mathcal{A}_1(L, \mathbf{pp}, x) \end{array}}$$

The sequentiality game is applied to an adversary $\mathcal{A} := (\mathcal{A}_0, \mathcal{A}_1)$. The adversary wins the game if $y_{\mathcal{A}} = y$ where $(y, \pi) := \textit{Eval}(\mathbf{pp}, x)$.

Sequentiality

For functions $\sigma(t)$ and $p(t)$, the VDF is (p, σ)-sequential if **no** pair of randomized algorithms $\mathcal{A}_0$, which runs in total time $O(\textit{poly}(t, \lambda))$ and $\mathcal{A}_1$ which runs in parallel time $\sigma(t)$ on at most $p(t)$ processors, can win the sequential game with probability greater than $\textit{negl}(\lambda)$

Some discussion

- t must be sub-exponential in λ.
- The reason is that the adversary can run in time at least t, if t is exponential in λ, then the adversary might be able to break the underlying computational security assumption.
- Parallelism in *Eval*.
- For a decodable VDF it is necessary that $|\mathcal{Y}| > \textit{poly}(t)$, and thus the challenge inputs to *Eval* have size $\textit{poly}(\log(t))$. Therefore, the algorithm *Eval* has up to $\textit{poly}(\log(t))$ parallelism.
- σ-sequentiality implies that min-entropy is $\Omega(\log(\lambda))$
- If t is sub-exponential in λ, then the output has $o(\lambda)$ min-entropy.

Value of $\sigma(t)$

- Value of $\sigma(t)$.

- Any candidate construction trivially satisfies $\sigma(t)$-sequentiality for some σ (e.g. $\sigma(t) = 0$). Security becomes more meaningful as $\sigma(t) \to t$. No construction can obtain $\sigma(t) = t$ by *Eval* design.
- An almost-perfect VDF would achieve $\sigma(t) = t - o(t)$ sequentiality. $\sigma(t) = t - \epsilon t$ sequentiality for small ϵ is sufficient for most applications.

Weaker VDFs

For small values of t, it may be practical for anyone to use up to $O(t)$ parallelism. So a weaker variant of VDF allows additional parallelism in *Eval*.

Weaker VDF

A VDF $V = (\textit{Setup}, \textit{Eval}, \textit{Verify})$ is a weak-VDF if it satisfies VDF definitions with exception that *Eval* is allowed up to *poly*(t, λ) parallelism.

Note that (p, σ)-sequentiality can only be meaningful for a weak-VDF if *Eval* is allowed strictly less than $p(t)$ parallelism, otherwise the honest computation of *Eval* would have more parallelism than adversary.

Construction of VDFs

Sequential function

(t, ϵ)-Sequential function

$f : \mathcal{X} \to \mathcal{Y}$ is a (t, ϵ)-sequential function if for $\lambda = O(\log(|\mathcal{X}|))$, the following conditions hold:

- There exists an algorithm that for all $x \in \mathcal{X}$ evaluates f in parallel time t using *poly*$(\log(t), \lambda)$ processors.
- For all $\mathcal{A}$ that run in parallel time strictly less than $(1 - \epsilon) \cdot t$ with *poly*(t, λ) processors:

$$\Pr\left[y_{\mathcal{A}} = f(x) \;\middle|\; y_{\mathcal{A}} \leftarrow \mathcal{A}(\lambda, x), x \overset{\$}{\leftarrow} \mathcal{X}\right] < \textit{negl}(\lambda)$$

Iterated Sequential Function

Iterated Sequential Function

Let $g : \mathcal{X} \to \mathcal{X}$ be a function which satisfies (t, ϵ)-sequentiality. A function $f : \mathbb{N} \times \mathcal{X} \to \mathcal{X}$ defined as $f(k, x) = g^{(k)}(x) = \underbrace{g \circ g \circ \cdots \circ g}_{k \text{ times}}$ is called an iterated sequential function if for all $k = 2^{o(\lambda)}$, the function $h : \mathcal{X} \to \mathcal{X}$ such that $h(x) = f(k, x)$ is $(k \cdot t, \epsilon)$-sequential.

Instances

- A chain of a secure hash function (like SHA-256).
- Exponentiation in a finite group of unknown order, the round function is squaring in the group.

Iterated Sequential Function

Assumption

For all $\lambda \in \mathbb{N}$, there exists an ϵ, t with $t = \textit{poly}(\lambda)$ and a function $g_\lambda : \mathcal{X} \rightarrow \mathcal{X}$ $s.t.$ $\log(|\mathcal{X}|) = \lambda$ and $\mathcal{X}$ can be sampled in time $\textit{poly}(\lambda)$ and g_λ is a (t, ϵ)-sequential function, and the function $f : \mathbb{N} \times \mathcal{X} \rightarrow \mathcal{X}$ with round function g_λ is an iterated sequential function.

- An iterated sequential function is sequential by definition and the trivial function uses only $\textit{poly}(\lambda)$ parallelism.
- However, the fastest generic verification is to simply recomputes the function, which do not satisfy the efficient verification requirement of VDF.

SNARK

SNARK

Let $\mathcal{L}$ denote an NP language with relation $R_{\mathcal{L}}$, where $x \in \mathcal{L}$ iff $\exists w \; R_{\mathcal{L}}(x, w) = 1$. A SNARK system for $R_{\mathcal{L}}$ is a triple of polynomial time algorithms $(\mathit{SNKGen}, \mathit{SNKProve}, \mathit{SNKVerify})$ that satisfy the following properties:

Completeness

$\forall (x.w) \in R_{\mathcal{L}},$

$$\Pr\left[\mathit{SNKVerify}(vk, x, \pi) = 0 \;\middle|\; \begin{array}{l} (vk, ek) \overset{\$}{\leftarrow} \mathit{SNKGen}(1^{\lambda}) \\ \pi \leftarrow \mathit{SNKProve}(ek, x, w) \end{array}\right] = 0$$

Succinctness

The length of a proof and complexity of *SNKVerify* is bounded by $\mathit{poly}(\lambda, \log(|y| + |w|))$.

SNARK

Knowledge extraction

For all adversaries $\mathcal{A}$ running in time $2^{o(\lambda)}$, there exists an extractor $\mathcal{E}_{\mathcal{A}}$ running in time $2^{o(\lambda)}$ such that for all $\lambda \in \mathbb{N}$ and all auxiliary inputs z of size *poly*(λ):

$$\Pr\left[\begin{array}{r|l} \mathit{SNKVerify}(vk, x, \pi) = 1 & (vk, ek) \xleftarrow{\$} \mathit{SNKGen}(1^{\lambda}) \\ R_{\mathcal{L}}(x, w) \neq 1 & (x, \pi) \leftarrow \mathcal{A}(z, ek) \\ & w \leftarrow \mathcal{E}_{\mathcal{A}}(z, ek) \end{array}\right] < \mathit{negl}(\lambda)$$

Impractical VDF from SNARGs

Consider the following construction for a VDF from a (t, ϵ)-sequential function f:

- Let $\mathbf{pp} = (ek, vk) = \textit{SNKGen}(\lambda)$ be the public parameter of a SNARG scheme for proving membership in the language of pairs (x, y) such that $f(x) = y$.
- On input $x \in \mathcal{X}$, the *Eval* computes $y = f(x)$ and a succinct argument $\pi = \textit{SNKProve}(ek, (x, y), \perp)$, and outputs $((x, y), \pi)$.
- On input $((x, y), \pi)$, the verifier checks $y = f(x)$ by checking $\textit{SNKVerify}(vk, (x, y), \pi) = 1$

Downsides

- Computing a SNARG is more than 100000 times more expensive than evaluating the underlying computation.
- The construction does not come asymptotically close to the sequentiality of the underlying computation f.

Incremental Verifiable Computation (IVC)

IVC

The basic idea of IVC is that for every step of computation, a prover can produce a proof of current state of computation. This proof is updated after every step of the computation to produce a new proof.

Properties

- The complexity of each proof in proof size and verification cost is bounded by $poly(\lambda)$ for any sub-exponential length computation.
- The complexity of updating the proof is independent of the total length of the computation.

Tight IVC for iterated sequential functions

Tight IVC

Let $f_\lambda : \mathbb{N} \times \mathcal{X} \to \mathcal{X}$ be an iterated sequential function with round function g_λ having (t, ϵ)-sequentiality. An IVC system for f_λ is a triple of polynomial time algorithms (*IVCGen*, *IVCProve*, *IVCVerify*) that satisfy the following properties:

Completeness

$\forall x \in \mathcal{X}$,

$$\Pr\left[\mathit{IVCVerify}(vk, x, y, k, \pi) = 1 \;\middle|\; \begin{array}{l} (vk, ek) \overset{\$}{\leftarrow} \mathit{IVCGen}(\lambda, f) \\ (y, \pi) \leftarrow \mathit{IVCProve}(ek, k, x) \end{array}\right] = 1$$

Succinctness

The length of a proof and the complexity of *SNKVerify* is bounded by $\mathit{poly}(\lambda, \log(k \cdot t))$

Tight IVC

Soundness

For all algorithms $\mathcal{A}$ running in time $2^{o(\lambda)}$

$$\Pr\left[\begin{array}{r} \mathit{IVCVerify}(vk, x, y, k, \pi) \\ f(k, x) \neq y \end{array} \middle| \begin{array}{l} (vk, ek) \xleftarrow{\$} \mathit{IVCGen}(\lambda, f) \\ (x, y, k, \pi) \leftarrow \mathcal{A}(\lambda, vk, ek) \end{array}\right] < \mathit{negl}(\lambda)$$

Tight Incremental Proving

There exists a k' such that for all $k \geq k'$ and $k = 2^{o(\lambda)}$, *IVCProve*(ek, k, x) runs in parallel time $k \cdot t + O(1)$ using *poly*(λ, t)-processors.

Existance of tight IVC

Tight IVC can be constructed from existing SNARK constructions.

VDF construction from IVC

Let $f_\lambda : \mathbb{N} \times \mathcal{X}_\lambda \to \mathcal{X}_\lambda$ defined by $f_\lambda(k, x) = g_\lambda^{(k)}(x)$, where g_λ is a (s, ϵ)-sequential function on an efficiently sampleable domain of size $O(2^\lambda)$. Given a tight IVC proof system $(\mathit{IVCGen}, \mathit{IVCProve}, \mathit{IVCVerify})$ for f, we can construct a VDF that satisfies $\sigma(t)$-sequentiality for $\sigma(t) = (1 - \epsilon) \cdot t - O(1)$

- *Setup*(λ, t): Let f_λ be an iterated sequential function and g_λ is the corresponding round function that is (t, ϵ) sequential. Run $(ek, vk) \overset{\$}{\leftarrow}$ *IVCGen*(λ, f_λ). Set k to be the largest integer such that *IVCProve*(ek, k, x) takes time less than t. Output $\mathbf{pp} = ((ek, k), vk)$.
- *Eval*$((ek, k), x)$: Run $(y, \pi) \leftarrow$ *IVCProve*(ek, k, x), output (y, π).
- *Verify*$(vk, x, (y, \pi))$: Run and output *IVCVerify*(vk, x, y, k, π).

Practical VDF Construction

VDF Construction

Setup

The setup algorithm *Setup*(λ, T) outputs two objects:

- A finite abelian group $\mathbb{G}$ of unknown order.
- An efficient computable hash function $H : \mathcal{X} \to \mathbb{G}$ that can be modeled as a random oracle.

Evaluation

The evaluation algorithm *Eval*$(\mathbf{pp}, x)$ is defined as:

- compute $y \leftarrow H(x)^{(2^T)} \in \mathbb{G}$ by computing T squaring in $\mathbb{G}$ starting with $H(x)$.
- compute the proof π as described later.
- output (y, π).

More about the construction

- In practical one might set $T = 2^{30}$.
- Let $g := H(x) \in \mathbb{G}$ be the basic element given as input to the VDF evaluator.
- Let $h := y \in \mathbb{G}$ be the purported output of the VDF, namely $h = g^{(2^T)}$.
- We need a succinct interactive argument for the language

$$\mathcal{L}_{EXP} := \left\{ (\mathbb{G}, g, h, T) : h = g^{(2^T)} \text{ in } \mathbb{G} \right\}.$$

VDF Construction (Wesolowski)

Verification

Given a tuple $(\mathbb{G}, g, h, T)$ as input, the prover and verifier run the following protocol to prove that $h = g^{(2^T)}$ in $\mathbb{G}$. Let *Prime*(λ) be the set containing the first 2^λ primes.

0. The verifier checks that $g, h \in \mathbb{G}$ and output $reject$ if not,
1. The verifier sends to the prover a random prime l sampled uniformly from *Prime*(λ),
2. The prover computes $q, r \in \mathbb{Z}$ such that $2^T = ql + r$ with $0 \leq r < l$, and sends $\pi \leftarrow g^q$ to the verifier,
3. The verifier computes $r \leftarrow 2^T \mod l$ and outputs 1 if $\pi \in \mathbb{G}$ and $h = \pi^l g^r$ in $\mathbb{G}$.

VDF Construction (Wesolowski)

Non-interactive variant

- The prover first generates l by hashing $(\mathbb{G}, g, h, T)$ to an element of *Prime*(2λ).
- The prover computes $\pi \leftarrow g^q$ as in step 2 above, and outputs this $\pi \in \mathbb{G}$ as the proof.
- The verifier computes l as the same way as the prover and decides as step 3 above.

VDF Construction (Wesolowski)

Verifier efficiency

The verifier needs to compute $r \leftarrow 2^T \mod l$, which only takes $\log_2 T$ multiplications in $\mathbb{Z}_l$.

Prover efficiency

Because T is large, we cannot write out q as an explicit integer exponent. However we can compute $\pi = g^q$ in at most $2T$ group operations and constant space using the long-division algorithm.

1. $\pi \leftarrow 1 \in \mathbb{G}, r \leftarrow 1 \in \mathbb{Z}$
2. repeat T times:

   $b \leftarrow \lfloor 2r/l \rfloor \in \{0, 1\}$ and $r \leftarrow (2r \mod l) \in \{0, \ldots, l-1\}$

   $\pi \leftarrow \pi^2 g^b \in \mathbb{G}$
3. output π (this π equals to g^q)

VDF construction (Pietrzak)

Verification for $(\mathbb{G}, g, h, T) \in \mathcal{L}_{EXP}$

0. The verifier checks that $g, h \in \mathbb{G}$ and outputs $reject$ if not,
1. If $T = 1$ the verifier checks that $h = g^2$ in $\mathbb{G}$, outputs $reject$ or $accept$, and terminate.
2. If $T > 1$ the prover and verifier do:
   a. The prover computes $v \leftarrow g^{(2^{T/2})}$ in $\mathbb{G}$ and sends v to the verifier. The verifier checks that $v \in \mathbb{G}$ and outputs $reject$ and terminate if not. Next, the prover needs to convince the verifier that $h = v^{(2^{T/2})}$ and $v = g^{(2^{T/2})}$ which proves that $h = g^{(2^T)}$. Since they have the same exponent, they can be verified simultaneously by checking a random linear combination, namely that

   $$v^r h = (g^r v)^{(2^{T/2})} \text{ for a random } r \text{ in } \{1, \ldots, 2^\lambda\}$$

   The verifier and prover do so as follows.

VDF construction (Pietrzak)

Verification for $(\mathbb{G}, g, h, T) \in \mathcal{L}_{EXP}$

b. The verifier sends to the prover a random r in $\{1, \ldots, 2^{\lambda}\}$.

c. Both the prover and verifier computes $g_1 \leftarrow g^r v$ and $h_1 \leftarrow v^r h$ in $\mathbb{G}$.

d. The prover and verifier recursively engage in an interactive proof that $(\mathbb{G}, g_1, h_1, T/2) \in \mathcal{L}_{EXP}$, namely that $h_1 = g_1^{(2^{T/2})}$ in $\mathbb{G}$.

VDF construction (Pietrzak)

Non-interactive variant

The prover generates the challenge r for every level of the recursion by hashing the quantities $(\mathbb{G}, g, h, T, v)$ at that level, and appends v to the overall proof π. Hence the overall proof π contains $\log_2 T$ elements in $\mathbb{G}$.

VDF construction (Pietrzak)

Verifier efficiency

At every level of recursion the verifier does two exponentiation in $\mathbb{G}$ to compute g_1, h_1 for the next level. Hence the verifier needs $2\log_2 T$ exponentiations in $\mathbb{G}$.

Prover efficiency

The prover needs to compute v for every level of the recursion. We denote v_i, r_i for the values of v, r for the i-th recursion.

$$\begin{aligned} v_1 &= g^{(2^{T/2})} \\ v_2 &= g_1^{(2^{T/4})} = (g^{r_1} v_1)^{(2^{T/4})} = \left(g^{(2^{T/4})}\right)^{r_1} g^{(2^{3T/4})} \\ v_3 &= g_2^{(2^{T/8})} \\ &= \text{ a power product of four elements } g^{(2^{T/8})}, g^{(2^{3T/8})}, g^{(2^{5T/8})}, g^{(2^{7T/8})} \end{aligned}$$

VDF construction (Pietrzak)

Prover efficiency

There exists an efficient way to compute these v_i.

1. When the prover first computes $h = g^{(2^T)}$, it stores 2^d group elements $g^{(2^{(i \cdot T/2^d)})}$ for $i = 0, \ldots, 2^d - 1$.
2. These 2^d values will be used when the prover computes $v_1, \ldots, v_d$, which takes about 2^d small exponentiation in $\mathbb{G}$.
3. The prover computes the remaining elements $v_{d+1}, \ldots, v_{\log T}$ by raising $g_{d+1}, \ldots, g_{\log T}$ to the appropriate exponents. This step takes a total of $T/2^d$ multiplications in $\mathbb{G}$.

Hence, the total time to compute these v_i is about $2^d + T/2^d$, which is optimal when $d = \frac{1}{2}\log_2 T$. So the VDF output and the proof π can be computed in total time approximately $T + 2\sqrt{T}$.

Other constructions

content...