# Masters Project - Visual Hashes (temporary title)

Adrien Laydu

Sometime in 2022

## 1 Context [TO BE EXPANDED/CLARIFIED]

TrustID is an Identity Provider (IDP) that can have various uses, such as 2-Factor Authentication (2FA) or digital signatures. As such, TrustID has to offer ways to securely identify, verify, and authenticate users. To achieve this, TrustID has a strong onboarding procedure, either using a video onboarding process or by having users physically go to an onboarding office, as well as providing an authenticator application for the user's mobile phone. Once onboarded, a user that wants to access a service secured by TrustID enter their credentials ([USERNAME] and password) on the TrustID website. The TrustID server then sends a challenge to the authenticator application, which then uses the unlocking mechanism of the phone to authentify the user: we assume only the targeted user can unlock their phone. The widespread use of biometrics as a way of unlocking phones (rather than traditional passwords or PINs) would ideally improve the security of this method drastically. However, unlocking processes typically offer one of the weaker methods as a fallback. Therefore, TrustID relies only on the ownership of the phone for security.

That brings us to our security issue: at time of the internship, When the user validates a [transaction on their mobile phone (i.e press the "accept" button of the authenticator app), they have no way of being sure that the prompt on their phone corresponds to the actual transaction they wish to effectuate. This could lead to some vulnerabilities: for example, an impersonation attack is possible, as shown here (we take the example of digital signatures for illustration purposes):

Say an an attacker A wants to sign a document we will call $d_A$ while impersonating user U. U uses a weak password, which means A can log as U on the TrustID website.

- A logs as U on TrustID website.
- A waits for U to ask for a challenge for any document.
- When U asks for a challenge for some document $d_U$, A asks the server for a challenge for $d_A$.
- The server will then send two challenges to U's phone, one for $d_A$ and one for $d_U$.
- Since U is unable to distinguish between the two challenges, they validate whichever challenge comes first.
- If the challenge for $d_A$ came first, then the attacker successfully signed a document while impersonating U.

While not that impactful when used on digital signatures, the same attack could be used for other (much more sensitive) TrustID services, such as 2FA: the attack described above would make the second factor useless.

This project is aimed at solving the security issue presented above. The way we will achieve it is to present the user with a unique, recognizable fingerprint of the document they requested a challenge for. While this could be achieved with traditional hash functions, the fact that human users will have to compare fingerprints calls for a more user-friendly solution. In particular, we want the fingerprints to have the form of pictures, as we hypothesize they are less cumbersome to compare.

**Note:** in practice, A's and U's requests for challenges are augmented with metadata. Different metadata calls for different capabilities of A: a UserID would not call for anything more than assumed earlier. A timestamp (currently in use at time of writing) requires A to send their request simultaneously with U. A nonce would require A to guess the nonce. [AT IMPLEMENTATION TIME] We will see [later] how the choice of metadata alter the probability of a successful attack.

# 2 Visual Hashing

## 2.1 Definitions and notations

### 2.1.1 Pixels

Because we work on image representation, we will work with pixels. We write $\mathbb{P}$ the set of all pixels. We will often need separate categories of pixels: in particular, we write $\mathbb{P}_2 \subset \mathbb{P}$ the set of pixel that may only take two values (namely "white" and "black"). We write $\mathbb{P}_{\mathsf{RGB}} \subset \mathbb{P}$ the set of pixels that are encoded by three color channels (Red, Green, Blue), each channel having an integer value between 0 and 255.

### 2.1.2 Visual hash function

We define a visual hash function as a function $\{0,1\}^* \to \mathbb{P}^{m \times n}$ of the form:

$$\mathcal{H}(x;\lambda) = V(H(x;\lambda);\lambda)$$

Where $\lambda$ is a security parameter, $H : \{0,1\}^* \to \{0,1\}^\ell$ is a standard hash function (such as $\mathsf{SHA-256}$) and $V : \{0,1\}^\ell \to \mathbb{P}^{m \times n}$ is a mapping from bits strings to arrays of pixels. The values of $\ell, m, n$ depend on the value of $\lambda$.

Whenever it is not explicitly needed, we omit the security parameter $\lambda$ in the notation. We call $h := H(x)$ a *hash*, and $F := V(h)$ a *fingerprint*. We note a visual hash function as $\mathcal{H} = (H, V)$ or when contexts allows it, just $\mathcal{H}$.

### 2.1.3 Visual indistinguishability

[TO BE CLARIFIED / EXPANDED IN DEDICATED SECTION]

For two fingerprints $F_1, F_2 \in \mathbb{P}^{m \times n}$, we write $F_1 \approx_v F_2$ if $F_1$ is *visually indistinguishable* from $F_2$, that is, if a user thinks $F_1 = F_2$ by looking at them. Note that $F_1 = F_2 \implies F_1 \approx_v F_2$.

### 2.1.4 Properties of visual hash functions

We adapt traditional hash functions properties to visual hash functions as follows [NOT SURE IF USEFUL YET]. When we say that an attack is "infeasible", we mean that an efficient [DEFINE EFFICIENT - POLY-TIME] adversary cannot realistically perform the attack because of its complexity. The properties are as follows:

- *First pre-image resistance*: we say a visual hash function $\mathcal{H}$ is first pre-image resistant if given $F \in \mathbb{P}^{m \times n}$, it is infeasible to find $x \in \{0,1\}^*$ such that $\mathcal{H}(x) \approx_v F$.
- *Second pre-image resistance*: we say a visual hash function $\mathcal{H}$ is second pre-image resistant if given $y \in \{0,1\}^*$, it is infeasible to find $x \neq y \in \{0,1\}^*$ such that $\mathcal{H}(x) \approx_v \mathcal{H}(y)$.
- *Collision resistance*: we say a visual hash function $\mathcal{H}$ is collision-resistant if it is infeasible to find $x, y \in \{0,1\}^*$ such that $x \neq y$ and $\mathcal{H}(x) \approx_v \mathcal{H}(y)$.

In order to better categorize and analyze visual hash function, we define two additional properties:

- *Intermediate pre-image resistance*: we say a visual hash function $\mathcal{H} = (H, V)$ is intermediate pre-image resistant if given $F \in \mathbb{P}^{m \times n}$, it is infeasible to find $h \in \{0,1\}^\ell$ such that $V(h) \approx_v F$.
- *Visual injectiveness*: we say a visual hash function $\mathcal{H} = (H, V)$ is visually injective if for any $h_1, h_2 \in \{0,1\}^\ell$, $h_1 \neq h_2 \implies V(h_1) \not\approx_v V(h_2)$. Note that visual injectiveness implies that $V$ is injective, because $V(h_1) \not\approx_v V(h_2) \implies V(h_1) \neq V(h_2)$

# 3 Context of application

## 3.1 Protocol

We consider the protocol shown in fig. 1, in which a visual hash function $\mathcal{H} = (H, V)$ is used. On the server side, $\mathsf{md}$ denotes some metadata such as a sessionID, timestamp, nonce, etc. [TO BE EXPANDED IN IMPLEMENTATION SECTION].

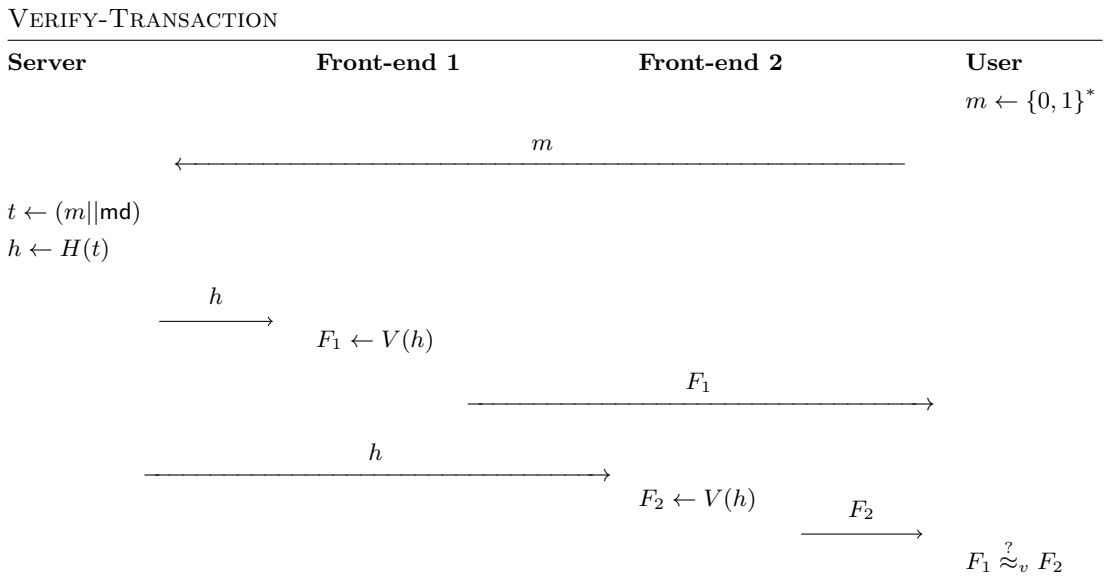


Figure 1: Protocol without adversary.

## 3.2 Threat models

### 3.2.1 A realistic TrustID threat

[GIVE IDEA OF TRANSACTION VALIDATION PROTOCOL (perhaps simplified version of Antoine's UML sequence diagram)]. Following [that], we assume an adversary cannot impersonate the TrustID server. However, an attack by an external adversary is still possible. We define, for an adversary $\mathcal{A}$, the game $\Gamma_{\mathsf{TrustID}}$:

$\mathsf{Game}\ \Gamma_{\mathsf{TrustID}}$

$m \leftarrow \{0,1\}^*$
$t \leftarrow (m||\mathsf{md})$
$h \leftarrow H(t)$
$F \leftarrow V(h)$
$\mathcal{A}(t^*) \rightarrow m'$
$t' \leftarrow (m'||\mathsf{md}')$
$h' \leftarrow H(t')$
$F' \leftarrow V(h')$
**return** $1_{F \approx_v F' \ \wedge \ m' \neq m}$

Where $t^*$ denotes some part of $t$: depending on the metadata we choose, $\mathcal{A}$ may know some bits of $t$ (e.g, the adversary knows at which time the transaction is made, or the user's ID). We assume however that the adversary doesn't know $m$. We consider our scheme achieves the minimal required security if for any efficient adversary $\mathcal{A}$, $\mathsf{Adv}_{\mathcal{A}}^{\Gamma_{\mathsf{trustid}}}(\lambda)$ is negligible in $\lambda$, where $\mathsf{Adv}_{\mathcal{A}}^{\Gamma_{\mathsf{trustid}}}(\lambda) = \Pr[\Gamma_{\mathsf{TrustID}} \text{ returns } 1]$. Figure 2 illustrates how the game translates to a concrete implementation.

While we could try and design some visual hash function that achieves this minimal security, we choose to go further and define a stronger threat model: this could alleviate our assumptions on the security of the rest of the TrustID architecture, as well as provide further insight. [Also it is fun]

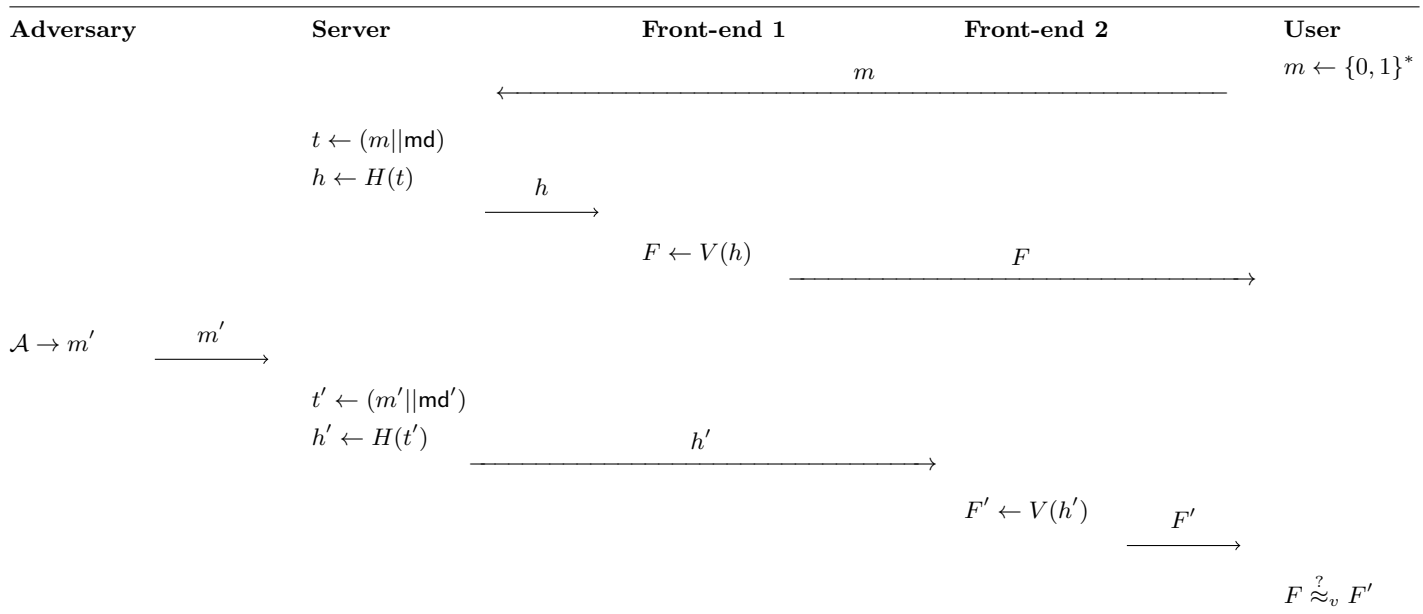


Figure 2: Protocol with a realistic TrustID adversary.

### 3.2.2 A stronger threat

We now define another game, with an adversary $\mathcal{A}$:

**Game $\Gamma$**

$m \leftarrow \{0,1\}^*$
$t \leftarrow (m||\mathsf{md})$
$h \leftarrow H(t)$
$F \leftarrow V(h)$
$\mathcal{A}(t, h, F) \rightarrow h'$
$F' \leftarrow V(h')$
**return** $1_{F \approx_v F' \ \wedge \ h' \neq h}$

In words, $\mathcal{A}$ wins if they can produce a hash $h'$, different from the true hash $h$, that yields a fingerprint $V(h')$ that is visually indistinguishable from the true fingerprint $F$. We will consider our scheme is secure if for any efficient adversary $\mathcal{A}$, $\mathsf{Adv}^{\Gamma}_{\mathcal{A}}(\lambda)$ is negligible in $\lambda$, where $\mathsf{Adv}^{\Gamma}_{\mathcal{A}}(\lambda) = \Pr[\Gamma \text{ returns } 1]$

We show that this security is indeed stronger that the minimal security we defined in the previous section. Concretely, we prove that, $\mathsf{Adv}^{\Gamma}_{\mathcal{A}}(\lambda)$ is negligible for all efficient adversaries $\implies \mathsf{Adv}^{\Gamma_{\mathsf{trustid}}}_{\mathcal{A}}(\lambda)$ is negligible for all efficient adversaries. We proceed by reduction: suppose $\mathcal{A}_{\mathsf{TrustID}}$ is an efficient adversary playing $\Gamma_{\mathsf{TrustID}}$ that wins with non-negligible probability. We build an adversary $\mathcal{A}$ playing $\Gamma$ that uses $\mathcal{A}_{\mathsf{TrustID}}$ to win as follows:

**$\mathcal{A}(t, h, F)$**

Extract $t^*$ from $t$ depending on what $\mathcal{A}_{\mathsf{TrustID}}$ needs
$\mathcal{A}_{\mathsf{TrustID}}(t^*) \rightarrow m'$
$t' \leftarrow (m'||\mathsf{md}')$
$h' \leftarrow H(t')$
**return** $h'$

Now, if $\mathcal{A}_{\mathsf{TrustID}}$ wins at $\Gamma_{\mathsf{TrustID}}$, then $\mathcal{A}$ wins at $\Gamma$. Therefore, $\mathsf{Adv}^{\Gamma}_{\mathcal{A}}(\lambda) \geq \mathsf{Adv}^{\Gamma_{\mathsf{trustid}}}_{\mathcal{A}_{\mathsf{TrustID}}}(\lambda)$, which was assumed to be non-negligible, which means $\mathsf{Adv}^{\Gamma}_{\mathcal{A}}(\lambda)$ is non-negligible. That concludes the proof: if there exists an efficient adversary

that breaks $\Gamma_{\mathsf{TrustID}}$, then there exists an efficient adversary that breaks $\Gamma$. This is equivalent to the statement we wanted to prove.

Figure 3 illustrate how the threat is stronger: the adversary can bypass the server to submit a hash directly to the frontend.

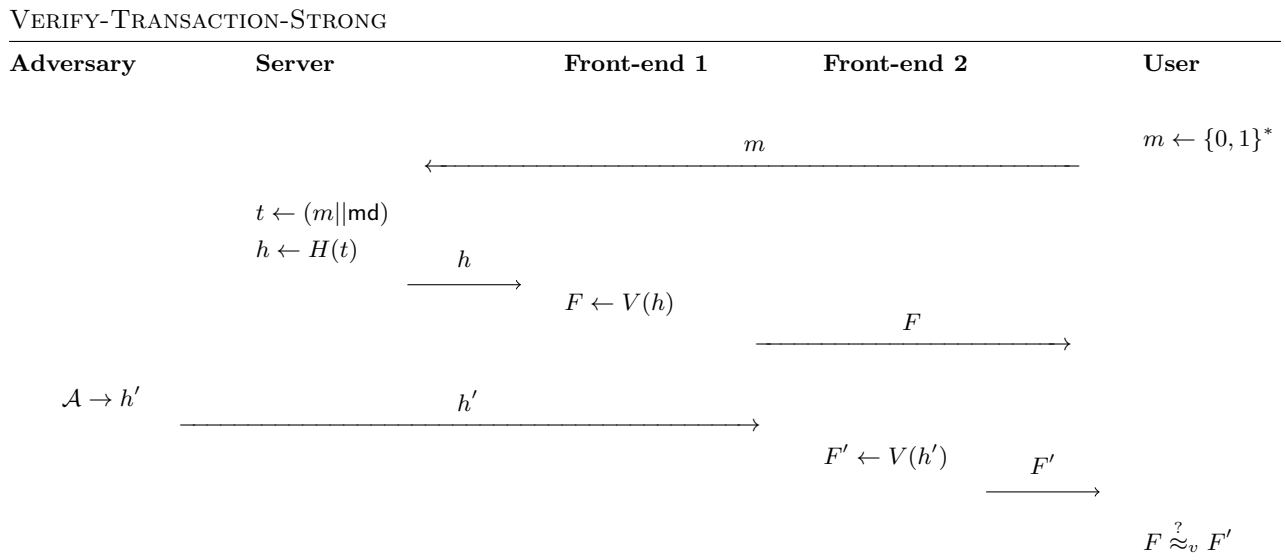


Figure 3: Protocol with a stronger adversary.

# 4 Security implications of visual hash function properties

We separate the different combinations of properties a visual hash function may have, to analyse their strengths and weaknesses in terms of security.

## 4.1 No visual injectiveness, no intermediate pre-image resistance

If a visual hash function $\mathcal{H}$ is neither intermediate pre-image resistant nor visually injective, the adversary can create a forgery:

1. From $F$, find $F' \neq F \in \mathrm{Im}(V)$ such that $F \approx_v F'$.
2. From $F'$, get $h'$ such that $V(h') = F'$.
3. Output $h'$.

This attack is efficient by assumption as long as the adversary can find $F'$ efficiently. Therefore we need at least one of the two proprieties to keep the adversary from making forgeries, or somehow make it infeasible to find $F'$ from $F$.

## 4.2 Visual injectiveness

If $\mathcal{H} = (H, V)$ is visually injective, then the adversary never wins: indeed, by definition of visual injectiveness, $h \neq h' \implies V(h) \not\approx_v V(h')$.

Moreover, visual injectiveness implies that the only case where $V(x_1) \approx_v V(x_2)$ with $x_1 \neq x_2$ is when $H(x_1) = H(x_2)$. That is, in order to find a collision on $\mathcal{H}$, the adversary must find a collision on $H$. Theferore the following implication holds:

$$\mathcal{H} \text{ is visually injective} \wedge H \text{ is collision-resistant} \implies \mathcal{H} \text{ is collision-resistant} \tag{1}$$

Since there are well-documented hash functions that are considered collision-resistant, visual injectiveness would be the ultimate goal for our hash function.

### 4.3 Intermediate pre-image resistance

If $\mathcal{H} = (H, V)$ is intermediate pre-image resistant but isn't visually injective, the attack described in section 4.1 is not feasible, because step 2 is infeasible by intermediate pre-image resistance. [BUT IS IT ENOUGH FOR SECURITY ?]

## 5 On visual indistinguishability

### 5.1 Evaluating the difference between fingerprints

We would like a metric to measure how well our scheme performs, so that we can analyse it formally. In particular, we need to define a distance between $d$ between two fingerprints $F_1, F_2 \in \mathbb{P}$ such that for some $\delta > 0$ (ideally as small as possible)

$$d(F_1, F_2) \leq \delta \iff F_1 \approx_v F_2$$

#### 5.1.1 normalised Hamming distance

A natural choice for evaluating the differences between binary images (i.e, that are in $\mathbb{P}_2^{m \times n}$) would be to compute the normalised Hamming distance of the two fingerprints

$$d_H(F_1, F_2) = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \mathbb{I}(F_1[i][j] \neq F_2[i][j])$$

where $F_1, F_2 \in \mathbb{P}^{m \times n}$, $(\cdot)[i][j]$ denotes the pixel at position $(i, j)$, and $\mathbb{I}$ is an indicator function. $d_H$ ranges between 0 and 1 included, the extrema being obtained by having two pictures that differ in zero or all of their pixels, respectively. This metric would be useful because it relates directly to the Hamming distance between of the hashes functions.

Unfortunately, it seems this distance does not achieve the properties we stated above. We can refute the right direction ( $\implies$ ) very easily. Indeed, fig. 4 shows two fingerprint with the smallest non-zero Hamming distance ($d_H = \frac{1}{mn}$, $mn = 256$) that are clearly visually distinguishible by humans. Concerning the other direction ( $\impliedby$ ), we should proceed with caution: shifting bits can produce images that look similar despite having a large Hamming distance. Figure 5 shows fingerprints having a normalised Hamming distance of $\frac{71}{256} = 27.73\%$ yet the results are not easy to distinguish.

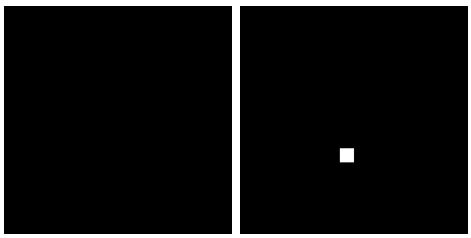

Figure 4: Two visually distinguishable fingerprints with low Hamming distance.

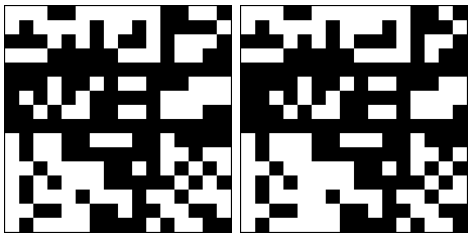

Figure 5: Two fingerprints that have a large Hamming distance, but are not trivial to distinguish.

### 5.1.2 HaarPSI distance

The Haar perceptual similarity index (HaarPSI) [ref] seems to be a relatively good measure to test whether a pair of fingerprints is distinguishable. The advantages it offers are a relatively low computation time compared to other PSI, which allows for automated parameter search, as well as a good correlation with human opinion scores [ref]. In contrast, the main downside is that although the results are re-linearized, the final output (a number between 0 and 1) is somewhat tricky to interpret: two clearly distinguishable images will have a HaarPSI around 0.1–0.2, and a HaarPSI over 0.6 yields a visual difference that is so small that we consider it imperceptible.

For our future scheme, we would of course like to minimize the HaarPSI distance between any 2 fingerprints. However, given the complexity of the HaarPSI formula, we cannot work our way from that to create a scheme. Instead, we pick a threshold value $T_{Haar} = [0.5]$ and we consider that

$$\mathsf{HaarPSI}(F_1, F_2) \geq T_{Haar} \implies F_1 \approx_v F_2$$

The value of $T_{Haar}$ was picked arbitrarily: indeed, we would ideally determine the value experimentally would require user testing, a process for which this project has neither the ambition nor the resources. Note that the condition on $T_{Haar}$ is necessary but not sufficient: indeed, we empirically found some pairs of fingerprints with a low PSI distance that are not easy to distinguish.

# 6 Existing schemes

## 6.1 Antoine's

Antoine's visual hash is a very simple scheme: $H$ is the standard SHA-256 function, and $V$ is a trivial bit-to-binary pixel mapping (a bit of value 1 gets mapped to a pixel of value 1 (white), a bit of value 0 gets mapped to a pixel of value 0 (black)). It is clear that this scheme is not intermediate pre-image resistant, as the trivial mapping can easily be reversed. Moreover, it is not sparse, as any combination of 256 binary pixels is a valid output: from a fingerprint $F$ one can produce a fingerprint $F'$ that differs from $F$ in only one pixel, which means $F \approx_v F'$ in general. Therefore, we can conduct the attack described in section 4.1 to create a forgery.

## 6.2 LifeHash

The LifeHash library [ref] employs the standard SHA-256 hash function, but it introduces some complexity in the definition of $b$ using John Conway's Game of Life [ref]. Let $\mathcal{G} : \mathbb{P}^{m \times n} \to \mathbb{P}^{m \times n}$ be the result of a single generation of the game of life rules on the input. We suppose fingerprints support element-wise addition, as well as multiplication by a constant.

Algorithm 1 describes the working of the LifeHash visual hashing function.

**Data:** $x \in \{0,1\}^*$
**Result:** $F \in \mathbb{P}^{m_n}_{\mathsf{RGB}}$
**begin**
  $B \longleftarrow \mathsf{Antoine}(x)$
  Initialize $G$ as the empty array
  **while** $|G| < 150$ ***and*** *Game of Life has not converged* **do**
    Append $B$ to $G$
    $B \leftarrow \mathcal{G}(B)$ ▷ One iteration of Game of Life rules, 1 is alive, 0 is dead
  Initialize $F$ as an all-white (255,255,255) fingerprint the same size as $B$
  **for** $i = 0, \ldots, |G| - 1$ **do**
    $F = F - G[i] * \frac{255i}{|G|}$ ▷ Aggregate all generations
  Apply RGB mapping to $F$, using some of the first bits to pick a color scheme.
  Apply symmetry to $F$, using some of the first bits to pick a transformation scheme.
  **return** $F$

**Algorithm 1:** LifeHash

Due to the unpredictable nature of the Game of Life, one would be tempted to think that this scheme is intermediate pre-image resistant: since one can construct a universal Turing machine in the Game of Life [ref], being able to predict for an arbitrary $h$ the final state of the Game of Life (or even whether the game of life converges in 150 generations)

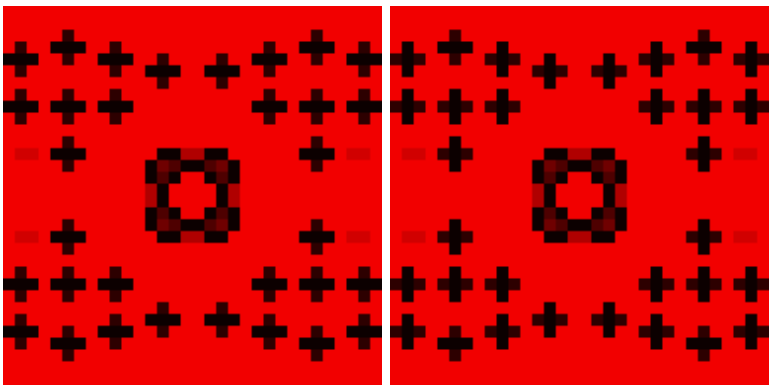

Figure 6: LifeHash outputs when $H = \hat{h}_1$ (left) and $H = \hat{h}_2$ (right)

can be reduced to the Halting problem [ref]. But while it may be intermediate resistant on complex inputs, the Game of Life has known, simple patterns that oscillate, which might cause problems if the input hash is simple (i.e. has a low Hamming weight). We describe an example of a forgery in order to illustrate that this scheme is neither intermediate pre-image resistant nor visually injective:

Let the initial state of the Game of Life $B_1$ contain a known oscillator. Let the following state sequence when applying the Game of Life rules successively:

$$B_1 \xrightarrow{\mathcal{G}} B_2 \xrightarrow{\mathcal{G}} B_3 \xrightarrow{\mathcal{G}} \ldots \xrightarrow{\mathcal{G}} B_1$$

Now, say we have two different input $x_1$ and $x_2$ such that $\mathsf{Antoine}(H(x_1)) = B_1$ and $\mathsf{Antoine}(H(x_2)) = B_2$. Then the execution of LifeHash will go through the same sequence of Game of Life (although rotated one state left for the second), yielding visually similar results. Figure 6 shows the output picture on input $x_1$ and $x_2$ when

$$h(x_1) := \hat{h}_1 = \texttt{0000000000000e00e0e0000e00000000eee00000000000006e030002000e0008}$$
$$h(x_2) := \hat{h}_2 = \texttt{0000000000004e4040444044000444404440444000000400640304020000e0008}$$

Both hashes contain the same period-3 oscillator (with the same initial state) in the center, and a collection of period-2 oscillators around it: 3 horizontally aligned pixels will flip to 3 vertically aligned pixels back and forth, thus giving a cross pattern on the fingerprint. Both need $\mathrm{lcm}(3, 2) = 6$ generations of the Game of Life to converge. The fingerprints differ only in the initial direction of the period-2 oscillators: on the left they start horizontally, whereas on the right they start vertically. Because each period-2 oscillators go through 3 entire cycles, the difference between the two setups becomes unnoticeable.

This pattern is really simple, but it is hard to determine whether other more complicated examples of oscillators combinations yield similar results.

## 6.3 PRG-based libraries

Some visual hash function use Pseudo-Random Generators (PRG) to create fingerprints. The idea is to seed a PRG with the input value, and let the PRG's randomness act to create fingerprints. Most implementations of such schemes use them directly on the messages (as opposed to hashes), which is not achievable in our case. However, they may have some desirable properties that we analyze in the following sections.

### 6.3.1 Blockies

Blockies [ref] is a light-weight javascript library used to create icons. It is at the time of writing quite popular (more than 1500 weekly downloads according to the official website), following the gain in popularity of cryptocurrency exchange and the need to identify 42-hexadecimal-characters Ethereum addresses.

Blockies is based on a custom-made PRG (!) based on the method used by Java to calculate hashCodes

**Data:** $x \in \{0,1\}^*$
**Result:** $F \in \mathbb{P}_{\mathsf{RGB}}^{m \times n}$
**begin**

$\mathsf{PRG}_{blockies}.\mathsf{Seed}(x)$ `// PRG_blockies returns values between 0 and 1`

$colorBack \longleftarrow (\mathsf{PRG}_{blockies}.\mathsf{next}() * 255, \mathsf{PRG}_{blockies}.\mathsf{next}() * 255, \mathsf{PRG}_{blockies}.\mathsf{next}() * 255)$
$colorFront \longleftarrow (\mathsf{PRG}_{blockies}.\mathsf{next}() * 255, \mathsf{PRG}_{blockies}.\mathsf{next}() * 255, \mathsf{PRG}_{blockies}.\mathsf{next}() * 255)$
$colorSpots \longleftarrow (\mathsf{PRG}_{blockies}.\mathsf{next}() * 255, \mathsf{PRG}_{blockies}.\mathsf{next}() * 255, \mathsf{PRG}_{blockies}.\mathsf{next}() * 255)$

Initialize $F$ as an all-zero fingerprint of size $m \times n$
**foreach** *pixel* $p$ *in* $F$ *such that* $p$ *is in the left half of* $F$ **do**
  $val \longleftarrow \mathsf{PRG}_{blockies}.\mathsf{next}() * 2.3$
  **if** $val < 1$ **then** `// happens with prob` $\frac{1}{2.3} \approx 43.5\%$
    $p \longleftarrow colorBack$
  **else if** $1 \leq val < 2$ **then** `// happens with prob` $\frac{1}{2.3} \approx 43.5\%$
    $p \longleftarrow colorFront$
  **else** `// happens with prob` $\frac{0.3}{2.3} \approx 13\%$
    $p \longleftarrow colorSpots$

Set the right half of $F$ to the mirrored left half of $F$ **return** $F$

**Algorithm 2:** PRG-based visual hash function

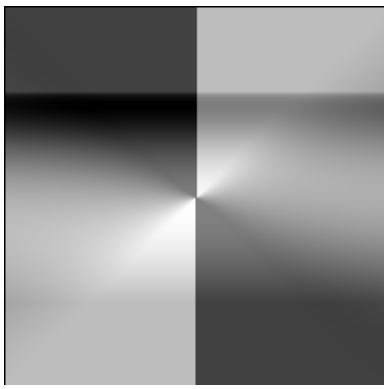

Figure 7: RandomArt visual hash.

### 6.3.2 RandomArt

[Ref] introduces another way of generating visual hashes using randomness: the idea is again to seed a PRG with the hash, then to use recursion to generate a random function to apply to each pixel. To visualise this, we construct an example. We choose a depth of 4, meaning we have 4 level of nesting. The following function was generated :

```
f(x,y) = SUB(SIN(SQRT(DIV(y,0.56))),SIN(ADD(DIV(y,x),SQRT(-0.35))))
```

In mathematical notation :

$$f(x,y) = \sin\left(\sqrt{\frac{y}{0.56}}\right) - \sin\left(\frac{y}{x} + \sqrt{-0.35}\right)$$

Note that each function maps the interval $[-1,1]$ to itself, using the necessary pre-processing if needed, hence the square root of the negative value will not cause any error.

This function is then evaluated at each pixel. Figure 7 shows the resulting fingerprint. In the true scheme, we repeat the process for each color canal.

The issue with this scheme is that the time needed to generate the random function grows exponentially with depth. Also, similarly to the Blockies scheme, the security of RandomHash relies entirely on the security of the PRG. Also, because of the choice of primitives and their properties, different functions can produce the exact same image; for instance

```
ADD(MULT(x,y), MULT(0.5, x)) == MULT(x, ADD(y, 0.5))
```

because $xy + 0.5x = x(y + 0.5)$ for all $x, y$ by distributivity of addition. However, [ref] gives us another tool: the way they evaluated if their hashes were recognisable by humans. Indeed, they give without proof two criteria to assess the quality of their image:

- The fingerprints should have a high compression factor, as low compression is a sign of a chaotic image.
- The fingerprints' frequency spectra should be concentrated around low frequences, as high frequences typically indicate noise.

The second criterion is what inspired us in the design of our scheme.

# 7 Fourier scheme

The complexity of the RandomHash function generation algorithm grows exponentially with depth, which can incur high computing time. So we design a new scheme with the following idea: instead of generating a random function and hoping that its frequency spectrum is centered around low frequences, we proceed "in reverse" and generate a frequency spectrum centered around low frequences then apply a 2-dimensional inverse Fourier transform to create the image.

## 7.1 2D Discrete Fourier Transform

The 2D Discrete Fourier Transform (2DDFT) of a function $f : (\mathbb{Z} \times \mathbb{Z}) \to \mathbb{C}$ is another function $\mathcal{F} : (\mathbb{Z} \times \mathbb{Z}) \to \mathbb{C}$ defined as follows:

$$\mathcal{F}(u, v) = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{-i2\pi(ux/M+vy/N)}$$

where $M$ and $N$ are the bounds of $f$. From $\mathcal{F}$, we can get $f$ back using the 2D Inverse Discrete Fourier Transform (2DIDFT):

$$f(x, y) = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} \mathcal{F}(u, v) e^{i2\pi(ux/M+vy/N)}$$

## 7.2 Spectrum generation

What we would like to achieve is to map a hash $h$ to an array of coefficients $S \in \mathbb{C}^{m \times n}$, and then apply the 2D inverse Fourier Transform on $S$ to obtain a fingerprint $F \in \mathbb{P}^{m \times n}$. In this section we detail the generation of $S$. We call $S$ a *spectrum*.

First, we notice that $S$ must be symmetric up to complex conjugation:

$$\begin{aligned}
\mathcal{F}(-u, -v) &= \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{-i2\pi(-ux/M-vy/N)} \\
&= \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{i2\pi(ux/M+vy/N)} \\
&= \mathcal{F}(u, v)^*
\end{aligned}$$

where $(\cdot)^*$ denotes the complex conjugate. Thus, we actually have to generate $MN/2$ pairs of complex conjugates in order to generate a $M \times N$ spectrum.

We want the spectrum to be concentrated around the axes. To achieve that, we will scale the coefficients so that the ones closer to whe axes have a greater module. Figure X shows a graphical representation of how we generate our spectrum. The darker squares represent a number with a greater module.

Once the spectrum is generated, we apply the 2DIDFT to get a 2D array of complex numbers.

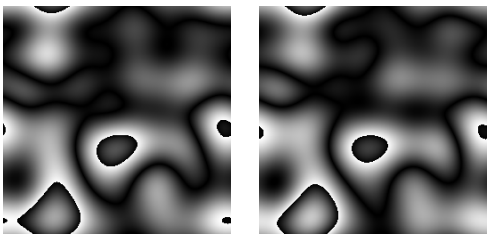

Figure 8: Two fingerprints with one bit flipped

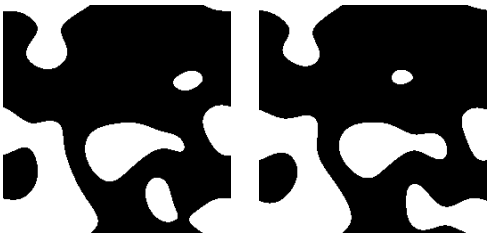

Figure 9: The two fingerprints, filtered

#### 7.2.1 Mapping bits to frequences

We would like every bit to have roughly the same "importance" with respect to the final image. Therefore, we map words of 4 bits to complex coefficients according to the encoding depicted on figure [FIGURE ENCODING], so that flipping one bit changes both the phase and the module of the resulting frequence. Each coefficient is then placed twice on the spectrum, such that the average distance to the origin of every word is roughly the same. [FIGURE SPECTRUM] shows where each coefficient is placed on the spectrum.

During testing, we noticed that the 2DIDFT tends to output mainly very small values (of the order of $10^{-4}$). Therefore, we normalize the spectrum by dividing each coefficient by the average module of spectrum. To get a the color value of the pixel at position $(x, y)$, we multiply the module of the coefficient at position $(x, y)$ with 255. (reducing modulo 256 if necessary).

Figure 8 shows the resulting fingerprints wits $h = \ldots$ and $h' = \ldots$. We can see that the results are very hard to distinguish. Furthermore, the PSI is 0.6367, which is more than the threshold value, therefore, we consider those hashes to be visually indistinguishable.

#### 7.2.2 Filtering

We notice that the high-contrast zones (that were produced by overflows) tend to stand out to the eye and thus help with the distinction of fingerprints [REF WOULD BE NICE]. Therefore, we apply a simple filter to enforce very high contrast zones: namely, from $S$, get a fingerprint $F$ by computing

$$F[x, y] = \begin{cases} 1 & \text{if } |S[x, y]| \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

We also slightly change the normalization of $S$: we perform an additional division by 2 on every coefficient such that the spectrum has now an average module of $\frac{1}{2}$. We do that to try and have black and white zones of roughly the same areas. Figure 9 shows the two same $h$ and $h'$. As expected, the two images are more easily distinguished. The PSI is 0.2135.

#### 7.2.3 Number of functions

While filtering offers a clear improvement, we are faced with an issue: we have to map $|h|$ bits to frequences. Standard hash lengths (128,256,512 bits) result in having to sample many coefficients, which will force us to sample coefficients too far away from the axes which translate to too much noise in the fingerprints. To address this problem, we perform the sampling procedure 3 times to generate 3 spectra $S_0, S_1, S_2$. Figure 10 shows the two same hashes when 3 functions are sampled. The current implementation of Fourier Hash supports 128-bit hashes.

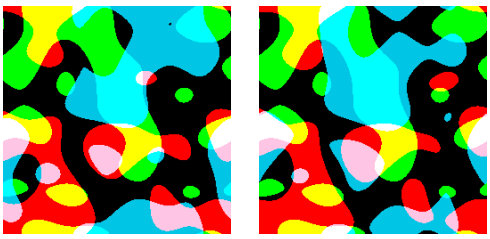

Figure 10: The two fingerprints, with 3 functions

| $S_0[x,y]$, $S_1[x,y]$, $S_2[x,y]$ | $F[x,y]$ |
|---|---|
| 0, 0, 0 | $Colors[0]$ |
| 0, 0, 1 | $Colors[1]$ |
| 0, 1, 0 | $Colors[2]$ |
| 0, 1, 1 | $Colors[3]$ |
| 1, 0, 0 | $Colors[4]$ |
| 1, 0, 1 | $Colors[5]$ |
| 1, 1, 0 | $Colors[6]$ |
| 1, 1, 1 | $Colors[7]$ |

Table 1: Color mapping

#### 7.2.4 Colors

We would like to improve the scheme by picking different set of colors for different hashes. We therefore have to define the way we assign a color to a pixel: we must choose $2^3$ colors to account for every possibility of the values of $S_0, S_1$ and $S_2$. Suppose we have an array named $Colors$ of length 8, containing rgb encodings.

We bow go further into details regarding how we construct the $Colors$ array. Once again, we would like every bit to have more or less the same impact on the resulting image. To do that, we will make use of two operations: modulos and additions modulo 2 (i.e XORs). First, we have to set the number of possible colors: we would like to have as many as possible as to maximize the diversity of the outputs. However, too much color possibilities would imply some of these colors are very close to each other up to the point where they are not distinguishable for a great part of the population. After testing different numbers, we chose to set the total number of colors to be 32.

We use a fixed 32-color array, which we call the *palette*. [ANNEX] shows the colors that make up the palette.

The first step we chose to implement is to rotate the palette, so that different hashes will have different set of colors. In order to compute the magnitude of the rotation, we compute $h \mod p_{\text{shift}}$, where $h$ is the hash interpreted as an integer. Once the palette is rotated, we have to pick colors indices. To do this, we perform an addition modulo 2 (XOR) on certain bits of the hash, which we call *parity bits*. Each bit of $h$ is a part of exactly one parity bit.

Let the parity bits $p_0$ to $p_{15}$ be defined as

$$p_i = \begin{cases} b_i \oplus b_{12+i} \oplus b_{24+i} \oplus \cdots \oplus b_{108+i} = \bigoplus_{k=0}^{9} b_{12k+i} & 0 \leq i < 12 \\ b_{108+i} \oplus b_{112+i} & 12 \leq i < 16 \end{cases}$$

We compute the colors indices:

$$c_j = p_{2j} || p_{2j+1} \text{ for } j \in \{0, \ldots, 7\}$$

where $||$ denotes concatenation, therefore $c_j \in \{0, 1, 2, 3\}$. Let $Palette$ be a fixed 32 colors palette. We fill $Colors$ as :

$$Colors[j] = Palette[4j + c_j] \text{ for } j \in \{0, \ldots, 7\}$$

Figure [FIGURE] depicts this process.

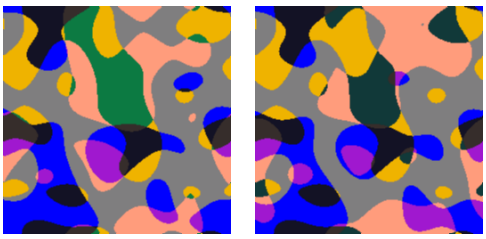

Figure 11: The two fingerprints, with color sampling

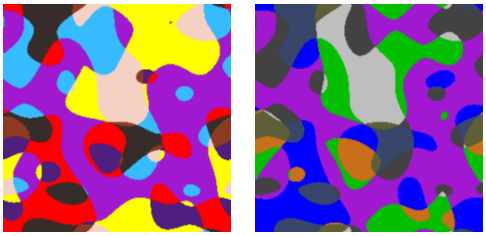

Figure 12: The two fingerprints, with the shift and permutation

#### 7.2.5 Color permutations

One thing we noticed during experimentation is that if two hashes have the same parity bits and same palette shift, then the colors will be the same and at the same place on the fingerprint. We introduce a permutation on the color array in order to address this issue. The way we compute the permutation is as follows. From [ref], we have a $(8, [4OR5])$-code, which means a code with words of length 8 such that the minimal distance between any two words is [4 OR 5]. We order arbitrarily the [2688 OR 616] words that make up this code. To get a permutation, we compute $j = h \mod [2683OR613]$ (which is prime). The permutation is the $j$-th word of the code.

Figure 13 shows the two fingerprints when the shift and permutations are performed. The PSI is 0.1810

#### 7.2.6 Symmetries

Humans tend to perform better at identifying symmetrical objects [ref]. Therefore, we would like our fingerprints to be symmetric. The way we do this is to first generate the fingerprint as we did previously, and then apply some symmetry according to a symmetry mode [DETAILS].

In order to determine the symmetry mode, we compute $h \mod 23$. The reason we chose this number is that is prime, small enough that we can implement different symmetry modes.

### 7.3 Collisions

We now analyse the different collisions against our scheme.

Let $h$ be a hash that produces fingerprint $F$, and $h' \neq h$ a hash that produces fingerprint $F'$. We define as collisions the following cases:

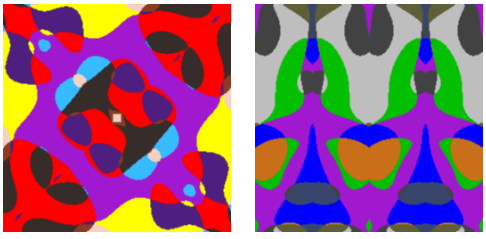

Figure 13: The two fingerprints, with the shift and permutation

- $h'$ and $h$ use the same symmetry mode $\iff h = h' \mod 23$
- $h'$ and $h$ use the same palette shift $\iff h = h' \mod p_{\text{shift}}$
- $h'$ and $h$ use the same colors (up to permutation) $\iff (h = h' \mod p_{\text{shift}}) \wedge (p_i = p'_i \, \forall \, i)$
- $h'$ and $h$ use the exact same colors $\iff (h = h' \mod p_{\text{shift}}) \wedge (p_i = p'_i \, \forall \, i) \wedge (h' = h \mod [2683OR613])$

The main idea [TO EXPAND/CLARIFY/PROVE] is that the more different two spectra are, the more different the fingerprints will look. That way, we can relate the difference between fingerprints with the Hamming distance between the hashes. To have the same the same symmetry, one would have to achieve $k - \ell = 0 \mod (|2| \mod q)$. Having the same palette shift AND same symmetry yields

$$\begin{cases} k - \ell = 0 \mod (|2| \mod p) \\ k - \ell = 0 \mod (|2| \mod q) \end{cases} \iff k - \ell = 0 \mod \text{lcm}(|2| \mod p, |2| \mod q)$$

Picking $p, q \in \{23, p_{\text{shift}}\}$ yields $\text{lcm}(|2| \mod p, |2| \mod q) = \text{lcm}(11, 28) = 308$. Therefore, if 2 bits are flipped, then either the palette is shifted differently, or the symmetry is not the same, or both. Either of these cases yields visually distinguishable results.

$$h + \sum_{\ell} 2^{\ell} - \sum_{k} 2^{k} = h \mod p$$

If only two bits are changed we have

$$\begin{aligned} h + 2^k - 2^{k+12m} &= h \mod p \\ h + 2^k(2^{12m} - 1) &= h \mod p \\ 2^k(2^{12m} - 1) &= 0 \mod p \\ 2^{12m} - 1 &= 0 \mod p \\ 2^{12m} &= 1 \mod p \\ 12m &= 0 \mod (|2| \mod p) \end{aligned}$$

Let us assume $n$ bits were flipped. Let us pick $p = p_{\text{shift}}, q = 23$. We have $pq = 667$, $|2| \mod p = 28, |q| \mod q = 11$

- $n = 1$ : The palette is flipped, the shift is different, the symmetry is different.
- $n = 2$ : Either the shift is different, the symmetry is different, or both.
- $n > 2, n$ odd : The palette is flipped.

For an even $n > 2$, there are collisions in the sense that some pairs of hashes use the same palette, same colors and same symmetry.

- For $n = 4$, we have $2^0 + 2^{84} + 2^{96} + 2^{108} \equiv 0 \mod 667$ which means flipping bits with indices $0 + d, 84 + d, 96 + d, 108 + d$ with $d \in \{0 \dots 11\}$ yields the same configuration if all the bits at these indices have the same value. We estimate the probability that $h$ contains a 4 bits of same value at at least one of the possibilities for $d$ to be $1 - \left(1 - \frac{1}{8}\right)^{12} = 1 - \left(\frac{7}{8}\right)^{12} \approx \frac{4}{5}$. (Note that this estimation assumes independent and uniform bits, which is not accurate but we just want to show that the probability has a high order of magnitude). Because that is a very high probability, we need patterns to change enough when 4 frequences are changed. Empirically, this seems to be the case