The Visual Computer manuscript No.
(will be inserted by the editor)

Direct Visualization of Cryptographic Keys for Enhanced Security

Oleg Lobachev

Received: date / Accepted: date

Abstract PGP public keys are relatively small binary data. Their hashes are used and also visualized for comparison and validation purposes. We pursue a direct, but previously unused approach. We produce colorful images of public keys and other binary data by generating drawing primitives from binary input. Optionally, we also include the hashes in the visualization. The visualization of raw data together with its hash provides a further security benefit. With it we can visually detect hash collisions.

The primary focus of this paper is a direct visualization of public keys. We tune the transparency heuristics for better results. Our method visually detects key spoofing on real SHA1 collision data.

Keywords Visualization · cryptography · public key · hash · collision · PGP · SHA1 · SHA2

1 Introduction

Public key cryptography is an important application in the modern world. Almost all communication channels in 2017 are encrypted using session keys that emerge from public-key-based key exchange mechanisms. PGP [27] was one of the front runners of using encryption in email exchange and pioneered the usage of military-grade encryption standards by the masses. Modern PGP implementation is GnuPG [10].

The typical way of obtaining a PGP public key is a key server or an initial message with the key attached. But how can one be sure that the key is correct and originates from intended source? The initial approach was to verify a *fingerprint* of a key [19]. Fingerprint is a cryptographic hash of a key with some metainformation added before hashing. Ideally, a fingerprint should be obtained from an alternative source (such as a phone call) and verified with the fingerprint of insecurely obtained key. However, fingerprints are quite tedious to transmit and to compare for a human. Efforts to improve this [3,17] produce either a small ASCII-art image or a “random art” of the fingerprint. For these visualizations following holds:

– They are visualizations of a *hash*,
– They are not quite aesthetic,

```
99 02 0d 04 55 5b 4b 79 01 10 00 a7 8e 75 e2 13
e6 65 86 a4 36 66 8e 76 9f 53 d3 51 fe d5 eb 58
3c ca 41 a9 db db 07 84 c8 ff 06 19 bc d1 be 2f
8f 67 2c 10 2e e4 0c 33 d5 fd fd 1f da 18 d6 75
35 6c 9e b0 aa 28 87 91 79 db f3 34 cc 73 a2 71
08 e9 3a 17 6c e1 e3 c0 af a3 3a 4c b2 39 cb c4
8e 3b 2f e3 e1 d1 2a 40 7a e8 de 9e d2 ac 12 e1
06 18 50 2a a6 c3 5b e5 87 ed 45 17 4c 49 39 6a
76 9b f0 db ff f2 88 ed 06 25 f2 36 3c 03 54 78
fd 8c 84 f5 69 ca c4 df c7 dc 82 4c ca 65 10 37
...
```

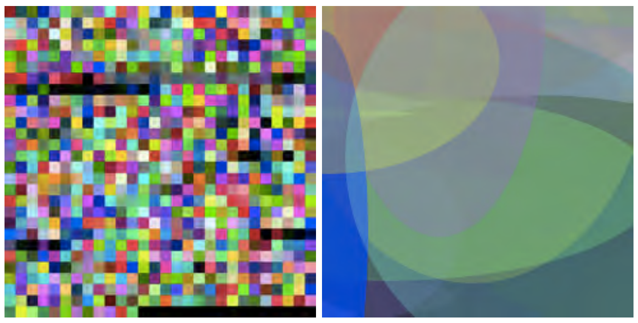

Figure 1 Interpreting author’s public PGP key (first few bytes are shown at the top) as a sequence of pixel colors is not aesthetically pleasant (bottom left), while interpreting the same data as a sequence of drawing statements (bottom right) produces nicer images.

O. Lobachev
Visual Computing, University Bayreuth, Universitätsstr. 30, 95440 Bayreuth, Germany
E-mail: oleg.lobachev@uni-bayreuth.de
`https://orcid.org/0000-0002-7193-6258`

– They can serve for a fast visual validation, as varying keys (even by few bits) should have very different hashes,
– They are prone to hash collisions if their hash function is compromised.

A very straightforward alternative mostly fails. We might try to interpret a key as a binary sequence of pixel colors. As the cryptographic data is mostly randomly distributed over the value range, the result looks mostly like random noise and is ill-suited for a visual comparison by a human, as Figure 1 shows.

For the best presentation of our method we provide a short introduction to public key cryptography and a notation of cryptographic protocols first and only then list the features of our contribution.

1.1 Formal notation for cryptographic protocols

To formally describe the interactions between multiple parties we use a simple notation similar to one by Cervesato et al. [4]. When Alice (A) sends Bob (B) a message, we write

$A \rightarrow B$: message contents.

An encrypted message is denoted with $\{\text{plain-text}\}_{\text{key}}$.

1.1.1 A typical PGP workflow

Alice (A) and Bob (B) want to communicate securely. Bob's public key is available at a public keyserver (S). Alice requests the key, the keyserver supplies Bob's key K_B and its fingerprint $f(K_B)$:

$A \rightarrow S$: request public key(B)
$S \rightarrow A : K_B, f(K_B)$.

Alice should have obtained the fingerprint $\hat{f}(\hat{K}_B)$ via a different channel, such as calling Bob per phone. If $\hat{f}(\hat{K}_B)$ and $f(K_B)$ are identical, the keyserver supplied a genuine key K_B. This step is, however, tedious and is often (and mistakenly!) omitted. Now Alice can use the key K_B to encrypt her plaintext message m to Bob:

$A \rightarrow B : \{m\}_{K_B}$.

Only Bob can decrypt $\{m\}_{K_B}$. The workflow is similar in many other cryptography applications.

1.1.2 A simple attack

Now we also have the attacker Eve (E), who manages to tamper with Alice's internet connection so Alice never reaches the genuine public keyserver S:

$A \rightarrow E$: request public key(B)
$E \rightarrow A : K_E, f(K_E)$.

Alice receives from Eve a malicious key K_E with a matching fingerprint instead of correct Bob's key K_B. If Alice never bothers to verify the fingerprint with Bob, she now sends a message that only Eve (still tampering with Alice's internet connection) can read. Eve intercepts the message:

$A \rightarrow X : \{m\}_{K_E}$.

Alice thinks X is Bob, but X is Eve. Eve can decrypt the message, as she has created the maliciously used public key K_E and knows the corresponding private key. Having the plaintext message m, Eve can even obtain the genuine Bob's public key K_B (in the manner described above) and play the man in the middle. Eve tampers the message to be m' and sends it:

$Y \rightarrow B : \{m'\}_{K_B}$.

Bob may think Y is Alice, but Y is in fact Eve. If Alice would verify the fingerprint, this would not happen. Our approach allows for more visual verification of public keys. Of course, more advanced techniques such as public key based signature would have prevented this very simple attack and would request more complicated approaches. However, the core idea is the same.

1.2 Contribution

In this paper we propose a visualization of short binary sequences (mostly public keys) that:

– Makes key collisions are visually detectable (Sections 3.4 and 4.1), thus improving security. Basically, the visualizations are *different* for two different keys with the same hash;
– Is aesthetically pleasant (as, e.g. Figs. 1 and 10 show);
– Is the same for same keys;
– Works for arbitrary binary data. Our method is independent from cryptographic algorithm that utilizes the keys we visualize.

2 Related work

To our knowledge such visualizations have not been attempted before. The closest approaches known to us are the hash visualizations: the work of Perrig and Song [17] and a the visualization used in `ssh` [3] (called `VisualHostKey`, Fig. 2, left). Our approach in same figure, right, visualizes the *key* directly. Perrig and Song [17] visualize only the *hashes* using the so-called "random art" technique. We deem our method more simple. In our opinion it also produces nicer images. Further, `VisualHostKey` is an ASCII-based visualization of hashes. We convert input bytes to a (potentially scalable) graphical visualization. Basically, where `VisualHostKey` has a printable subset of ASCII characters and their position,

```
+---[RSA 2048]----+
|XBoo+.           |
|X o=.            |
|.= .o            |
|+ o.o            |
| E =oo. S        |
|o *.B+..         |
|.+oOo+           |
|.==.+ .          |
|o=+.             |
+----[SHA256]-----+
```

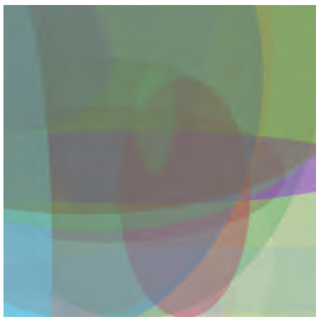

Figure 2 Visualizing `ssh` keys. Left: an existing hash-based key visualization in `ssh`, right: direct visualization of the key with our method.

we have coordinates and colors on the canvas. In a contrast to both Perrig and Song and `VisualHostKey`, we visualize the keys directly. It is possible to combine the key and the hash visualizations in our method. Awni [2] shows an interesting approach. There, faces are generated from hashes of the keys to provide better recognition. While one might argue, that people can better differentiate faces than abstract patterns, Awni still utilizes hash functions, while we use the public key and add, optionally, its hash.

Hash function attacks [26,25,12,21] are the reason why hash-only visualizations are vulnerable. In this paper we use SHA2 hashes for which no collisions are yet known. Generally, our approach can be used with any hash algorithm, but also completely without it. Our method facilitates a detection of a changed key even if the hash value is the same. We show this on a mock example (Fig. 8), but also on real SHA1 collision data (Fig. 12, data courtesy of Stevens et al. [21]). Counter-cryptanalysis [20] also detects possible collisions, as they typically rely on very special circumstances.

Loosely related to our method are graphical passwords [9,23]. Visual cryptography [13,11] is a completely different technique, where cryptographic methods are used to convey and securely share visual information. We, however, visualize cryptographic data.

Visualizations in other areas of computer security include [24,7,14]. Generally speaking, both Teoh et al. [24] and Conti et al. [7] visualize network traffic on a 2D canvas. Conti et al. [7] focus on data analysis. Features like typical packet size and other characteristics can be seen in their visualizations. Their goal is the detecting of reoccurring patterns and overall structure. We focus on generating memorable shapes—at least our user should notice if the visualization is suddenly different from how it always was, as detailed below. Teoh et al. [24] showcase an activity visualization in a network as a 2D plot. In their visualizations, e.g. an ongoing attack can be easily identified. They also mention further techniques and 3D visualizations, such as classical information visualizations of network activities. Our visualizations do not look at the network at large, but focus on a single most critical information during a concrete network session, the identity of the counterpart. We are not concerned with network connections at all in this paper and visualize the public keys as such.

Nataraj et al. [14] visualize malware as grayscale images. Such visualizations allow for similarity classifications. Basically, we allow our users to do the same with cryptographic keys. In contrast to our approach, they use a more sophisticated visualization technique. Further, Nataraj et al. use GIST features [15] in a post-mortem analysis, while in our application non-similarities are striking enough to be noticed with a naked eye. Our application strategy is to visualize cryptographic data during each session. The user would notice a quite different image (Fig. 8) if an attack is being executed. A yet another difference is that Nataraj et al. concentrate on malware payloads, somewhat large data, while cryptographic keys and accompanying data typically stay well under 10 KB. Precisely their small size allows us to visualize the data directly.

A long-time companion of cryptography in practical applications is steganography. It is not concerned with how to make a message unreadable to third parties, but with hiding the fact the message is present [22]. A connected issue is digital watermarking [8]. Typical targets are images, although there are methods to hide information in meshes [5,6].

3 Methods

3.1 Informal description

We interpret the input bytes as (parts of) vector graphics statements. For example, two bytes state the starting coordinates, two bytes state the end coordinates and four bytes state RGB color and alpha channel (Fig. 3). While it is possible to use an additional byte for the line thickness (Fig. 4), we decided against it. The final result of our “line” visualization is shown in Figures 3 and 9. This approach is much more aesthetically pleasant than the direct depiction of values (Figs. 1, 3). The visualization also shows how large the input is (Figs. 3 and 9), as its size in bytes influences the number of line strokes. Key size is a crucial information that is often needed, as it defines the grade of security. Keys with shorter key size (in the same cryptographic algorithm) might be compromised earlier, so they need to be replaced sooner.

Still, we were not yet satisfied with this visualization and changed a major detail. Instead of painting lines, we paint ellipses. In this way the first four bytes specify the parameters of the ellipse. The next three bytes define the color and the last, eighth byte defines the transparency. Basically this approach produces our visualizations in Figures 1 and 10. We provide more visualizations in the supplementary material (`https://dx.doi.org/10.5281/zenodo.817656`).

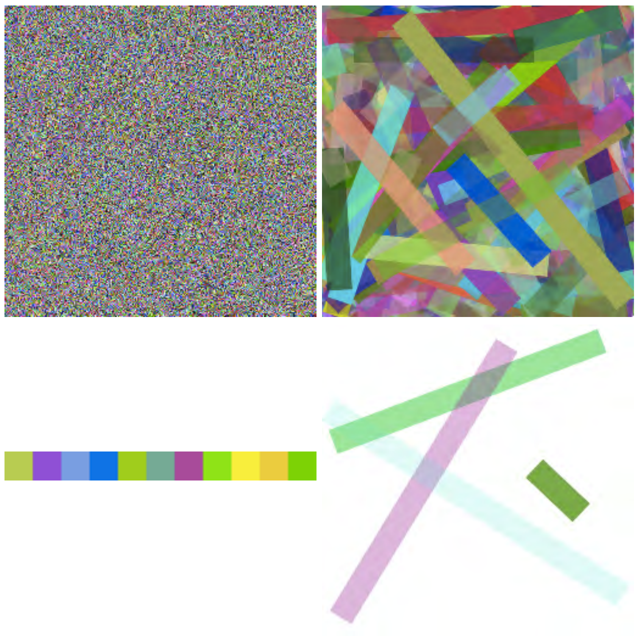
Figure 3 Visualizing 192 KB of random data directly (top line) and its SHA256 hash (bottom) as RGB pixels (left) and with our "line" method (right). Notice how our method shows the size of input data, while remaining in the same gist of visualization for data range from 32 to 192K bytes.

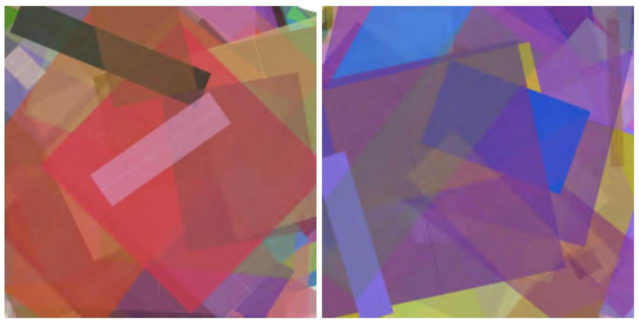
Figure 4 Here we also include line thickness into the set of varying parameters. For best results, here the transparency also should be adjusted, however this was not the case with these images.

3.2 Formal description

For our "line" method, in an 8-byte sequence $a, \dots, h$ we define

$$\mathit{from} = [a, g] \quad \text{and} \quad \mathit{to} = [b, h],$$

the line of constant thickness is drawn on a 256×256 canvas between points *from* and *to*. The bytes $c, \dots, f$ define directly RGBA color of said line. This method produces images in Figure 9—simple but already more visually pleasing than a direct pixel-based visualization (Fig. 3) or hash-based ASCII visualization (Fig. 2). We have also experimented with varying line thickness, but were less satisfied with these results (Fig. 4).

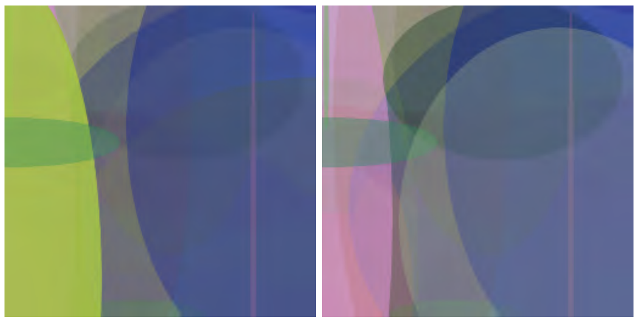
Figure 5 The transparency problem. The visualization left uses direct transparency values from the visualized key. On the right hand side the same key is visualized using adjusted transparency values.

For our "ellipse" method, in an 8-byte sequence $a, \dots, h$ we define on a 256×256 canvas an ellipse position and size using $a, \dots, d$, per

$$C = [a, b], \quad D_1 = c, \quad D_2 = d,$$

where C is the center of the ellipse, D_1 and D_2 are both diameters. The RGB color of the current ellipse is defined as e, f, g. Using the "real" transparency byte h as the input value t in the below procedure, we obtain the adjusted transparency value $\hat{t}$ (Section 3.3). This results in visualizations shown in Fig. 1, bottom right, Fig. 2, right, Fig. 5, right, Fig. 6, right, Figs. 10, and 11.

Possible future adjustments may include varying the drawing primitive or using segments of ellipses instead of permanent full 360° primitive.

3.3 Transparency adjustments

When we directly use transparency values from eighth byte of the block, it produces less nice images. Basically, it happens when the transparency is low and the ellipse size is high, as Figure 5, left shows. To adjust this, the final method uses a distance measure obtained from a not quite straightforward heuristics.

A direct approach would be to take the area of the ellipse, or a heuristics for that, and to increase the transparency for large area ellipses. We still want to factor in the last, eighth byte from the input into the transparency value. However, we found that such approach produces less visually appealing images (Fig. 6 compares with our actual method). The reason is clear: now absent larger opaque features yield a bleak, not very memorable images (Fig. 7). Hence, we follow a different, more involved, heuristics.

Our transparency heuristics uses the first four input bytes ("coordinates") and the "real" transparency value t (i.e. value of the eighth byte). It computes the adjusted transparency $\hat{t}$ from D:

$$\hat{t} = \left\lfloor \delta + \frac{st}{D} \right\rfloor \bmod T.$$

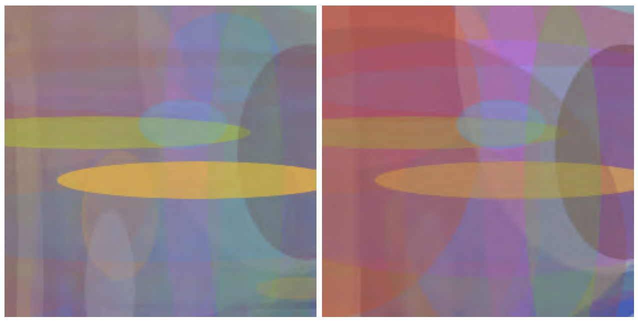
Figure 6 Adjusting transparency values. Left: a straightforward way with area heuristics, right: our final approach.

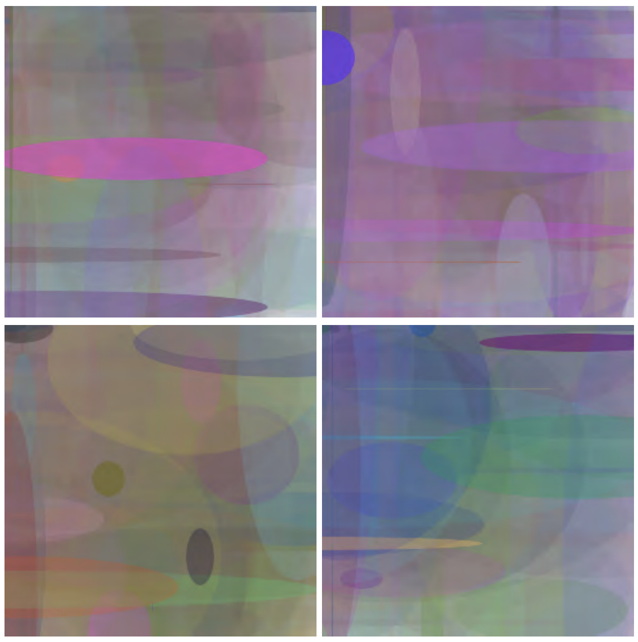
Figure 7 Examples when straightforward transparency is aesthetically bad.

In our implementation we used the constants $\delta = 10$, $s = 100$, and $T = 200$. We compute D as $|a-c|+|b-d|$ for first four bytes $a,\ldots,d$. This is in a sense a Manhattan norm. To give an example, for $D = 500$ and $t = 200$ the adjusted transparency value is $\hat{t} = 50$. Figure 5, right, Figure 6, right, and Figure 10 show visualizations with transparency values adjusted. Notice that this means a non-even distribution of transparency across the image, but the particular configuration presented was empirically found to be quite aesthetically pleasant. The more "linear" adjustment of transparency was visually less viable, as Figs. 6 and 7 show.

For the visualization the input is processed in eight byte sequences. The trailing number of bytes that is smaller as eight is discarded from visualizations shown here.

3.4 Benefit for security

Our method is suited to visualize *any* kind of binary data. Visualizing a key alone makes the visualization prone to subtle key changes that would be immediately detected with a cryptographic hash function, but might go undetected with a key-only visualization. Further, if a hash function is compromised, and collisions are easy to find, then key validation and visualization with these hashes is also compromised.

To combat this we visualize *both* the raw key and its SHA256 hash—a 256 bit version of the SHA2 family of hash functions [1]. Figure 8 showcases the result. If a key is changed in a manner undetected by our key visualization, the hash will be different (Fig. 8, top). For demonstration purposes we decoupled the key and hash visualizations and used the genuine key with a different hash. If the hash function is compromised and allows to create hash collisions easily, the key would be very different, even if the hash is the same. Fig. 8, left, shows this. Even with the same hash the actual keys differ a lot, which can be visually detected.

To circumvent such a combined visualization, the attacker needs to find hash collisions in a cryptographic hash function at will, while minimally changing the key bits (such that direct key visualizations look similar), and freely changing the last < 8 bits in the key that are currently omitted in our visualizations. The latter is circumvented easily by padding the key to a length divisible by eight before visualizing. Finding a collision that only minimally changes the the visualization output might be quite hard, as Figs. 9 and 11 showcase quite different images all the time. These figures show key-only visualizations of public keys from author's key chain.

We deem the attack a quite challenging task as mapping minor changes in the input to large changes in the output is the essence of a good hash function. The attacker would need to find a hash collision (i.e. a very different "fake" key with the same hash as genuine key) while maintaining a low bit-wise difference between the genuine key and the fake key.

Section 4.1 shows a real-world example using SHA1 collision data.

3.5 Protocol descriptions of prevented attacks

Next, let us regard two scenarios in which our visualizations would be of an immediate utility. We use the formalism from Section 1.1.

3.5.1 Weak hashes

Imagine, the hash function used for fingerprinting is compromised, but Alice never took notice of it. Eve tricks Alice into using a wrong key; Eve can intercept Alice's communication

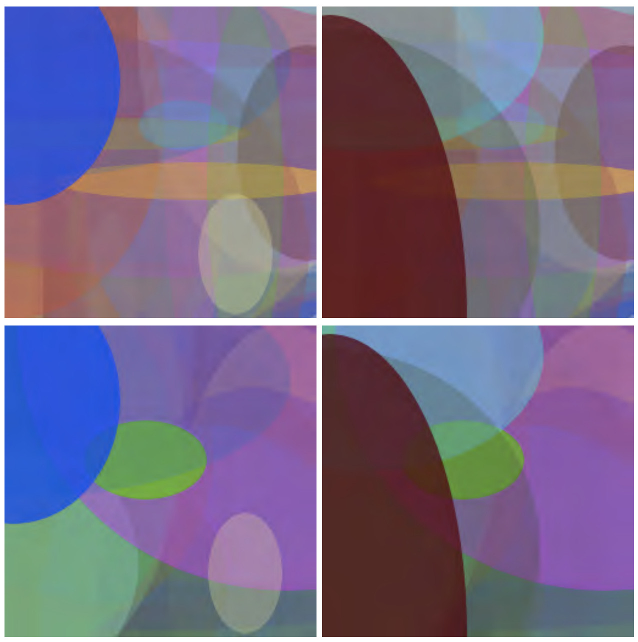

Figure 8 Showcasing how key spoofing can be detected with our method, a synthetic image. (Figure 12 shows real collision detection in SHA1.) All visualizations use the same method. Top left: Original PGP key with its genuine SHA256 hash. This is the initial reference for the user. Top right: We assume the key is rigged in a way that is not detectable in our visualization, but the hash has changed. This is strikingly visible. To produce this image, we have basically changed the hash, but not the key. Bottom left: Assume it was possible to find a hash collision, the hash is the same as the genuine hash. But the key itself has drastically changes to allow the collision, which is detected with our visualization. In other words, in case of key spoofing, a key with the same hash can be found, but the *key* visualization of the spoofed key differs. Bottom right: For completeness, a fake key with a hash that does not match the reference hash. (In this image we merely visualized a different key with *its* hash.)

to Bob.

A has Bob's key K_B and is aware of its fingerprint $f(K_B)$
E creates a fake key K_E with matching fingerprint.

The key K_E is maliciously generated in such a manner that fingerprint $f(K_E)$ is the same as $f(K_B)$. This is only possible because f is compromised.

E tricks A into using K_E instead of K_B.
A computes the fingerprint of K_X, it is valid.

The key K_X is what Alice believes is K_B, but it is really K_E. The fingerprint did not change. Now Alice sends a message using this key:

$A \to X : \{m\}_{K_X}.$

Eve can intercept the encrypted message and read it, because the key K_X is actually K_E. All key visualization systems based on the same fingerprint method f would fail, as the hash function at the heart of f is compromised. With our approach, the key is visualized directly and Alice can immediately see the difference (Fig. 8). Next, regard a more real-life example.

3.5.2 Online banking example

As of now, too many people are tricked into entering their online banking credentials at a wrong website. Illegitimate acquisition of credentials is called “phishing”. Thus the question “How can I allow the user to quickly verify that the public key of this online banking site is genuine and the same as it was on the real online banking site yesterday?” has, unfortunately, enough motivation. We regard this case next.

In a HTTPS session [18], a session key $K_{A,B}$ is negotiated. A *certificate* c, a combination of server public key and metainformation, such as domain name, is cryptographically signed during the initial setup of the service. To alleviate the validation procedure, a hierarchy of certificate authorities is used. Basically, a chain of “who trusts whom” is established, from the server certificate c up to a root certificate that is universally trusted. At a HTTPS session, the certificate of the website is checked whether it is still trusted. Here, we replace the hierarchy of certificate authorities with a stronger version: a direct query to the trusted third party. The problem is that in both cases the validity is established for the link website–entity that obtained the certificate. Bluntly, if a bank has a legitimate certificate for `bankname.com`, the attacker can obtain a *legitimate* certificate for `mybankname.net`. Both certificates are valid and legitimate. It is the user's responsibility to discern if the domain name is correct and whether the certificate was issued to the correct legal entity. The users, however, often make mistakes and trust wrong websites.

Alice (A) is going to visit her online banking site (B). Their communication is encrypted by a session key $K_{A,B}$. The correctness of the communication can be verified by checking the certificate c of the website with a trusted third party (T). The certificate is basically a public key with metadata. The correct behavior is:

$A \to B : \{\text{access website}\}_{K_{A,B}}$
$B \to A : \{\text{login form, } c, f(c)\}_{K_{A,B}}$
$A \to T : \{\text{is } c \text{ valid?}\}_{K_{A,T}}$
$T \to A : \{\text{our fingerprint for } \hat{c} \text{ is } \hat{f}(\hat{c})\}_{K_{A,T}}$
A compares $f(c)$ and $\hat{f}(\hat{c})$ and they are identical
$A \to B : \{\text{credentials for money transfer}\}_{K_{A,B}}.$

Notice that we do not automatically verify if the certificate c belongs to the bank B, but if it is a correct, valid certificate. An attacker might also own such a certificate, of course for a different website. She can lure Alice to it, while pretending to be the bank, as we will see below.

A is lured to a website E, thinking it is B

$A \to E : \{\text{access website}\}_{K_{A,E}}$

$E \to A : \{\text{login form}, c', f(c')\}_{K_{A,E}}$

$A \to T : \{\text{is } c' \text{ valid?}\}_{K_{A,T}}$

$T \to A : \{\text{our fingerprint for } \hat{c}' \text{ is } \hat{f}(\hat{c}')\}$.

A compares $f(c')$ and $\hat{f}(\hat{c}')$ and they are identical.

Notice that all Alice knows is that the certificate is valid. It is her responsibility to check that c' is actually a certificate of her bank B. If Alice has failed to notice the counterpart E she is talking to is not the bank:

$A \to E : \{\text{credentials for money transfer}\}_{K_{A,E}}$.

A looses all her money to E.

If Alice would have compared our visualization of the certificate c with the visualization of malicious c', she would have noticed the fact of an attack. The point is to recognize that the malicious website uses a different certificate c', as valid as it might be. Now, online banking is a quite routine operation. If the original visualization is displayed every time Alice uses online banking, we speculate that she would remember it after few sessions. Our visualizations are quite distinctive to be easily remembered and discerned by humans (Figs. 9–11).

The bank might even supply the visualization on a different channel, so Alice does not need to remember the genuine certificate image. For example, the visualization of genuine online banking certificate could be placed on the banking card. Of course, Alice would also notice every goodwilled change of the certificate, and hence her card would need to be re-issued every time the certificate changes. This is a distinct issue, however.

4 Results and Discussion

We have visualized keys from author's key ring with lines in Figure 9. Figure 10 shows the same keys, visualized using ellipses with adjusted transparency. In this case we add the SHA256 [1] hash to the visualized data. We utilized the SHA256 implementation from Open SSL [16]. All images are quite different, it is easy for a human to distinguish them. In most cases the visualizations are even quite artistic.

4.1 SHA1 Collision

The weaknesses of SHA1 hash algorithm have been known for more than a decade [25]. Recently, the first actual collision became publicly available [21]. As lined out in Section 3.4, our visualization should reveal key spoofing based on hash collisions. Hence, we have tried our approach on the collision example with SHA1 hashes. Notice that everywhere else in this work we use SHA2 hashes. For this hash function no practical collisions are known. Available files[1] with collision are small PDF files. We have produced a series of images from 512 byte chunks at same offsets in the respective files. We additionally compute and visualize in the same image a SHA1 hash of each chunk. (This does not help, as the hashes suffer from the collision and are identical.) Both files are mostly identical, with a smaller sequence with the collision that is different in respective files. These sequences have however the same SHA1 hash [21]. Figure 12 shows that in our visualization said differences are detected despite the collision. This distinguishes our system from *all* other methods used for visualization of public keys, as all these methods utilize hashes.

4.2 Discussion

Should our visualization provide *similar*, but subtly different images for varying keys, these keys are also similar in their byte-wise values. This is a corollary from using a direct visualization and requiring same visualizations for same keys. For production use, however, we augment the key visualization with a subsequent hash visualization. Now we obtain different visualizations also for similar keys. We are able to visually detect the key spoofing, as the visualizations are *different* for two different keys with the same hash.

The method is specifically geared towards small files, such as public keys. However, it makes also little sense to visualize a large file directly, rather interesting is its hash. Au contraire, in public keys both hash and direct visualizations are viable. Hash visualization contributes the cryptographic information: keys that differ only in few bits should be considered as radically different. Hash collisions, though, would deem two radically different keys as the same. Our method (in key+hash mode) shows similar (but varying, because of hash) visualizations for similar keys. Our method cannot fall into the pitfall of an eventual hash collision as not *only* hashing is used. In the actual key+hash visualization, two kinds of attacks can be detected. A "visual" key spoofing attack—that does not exist yet—can be detected via depiction of varying hash values of an uncompromised hash function. A hash collision (stemming from usage of a compromised hash function) can also be detected via direct key visualization. We model these two situations in Figure 8.

Compared to Perrig and Song [17] we provide a nice and potentially scalable graphics by *directly* painting the key (and, eventually, additional data). Our method is more direct, this conveys more security. The `VisualHostKey` visualiza-

[1] SHA1 collision files were obtained from `http://shattered.it/`.

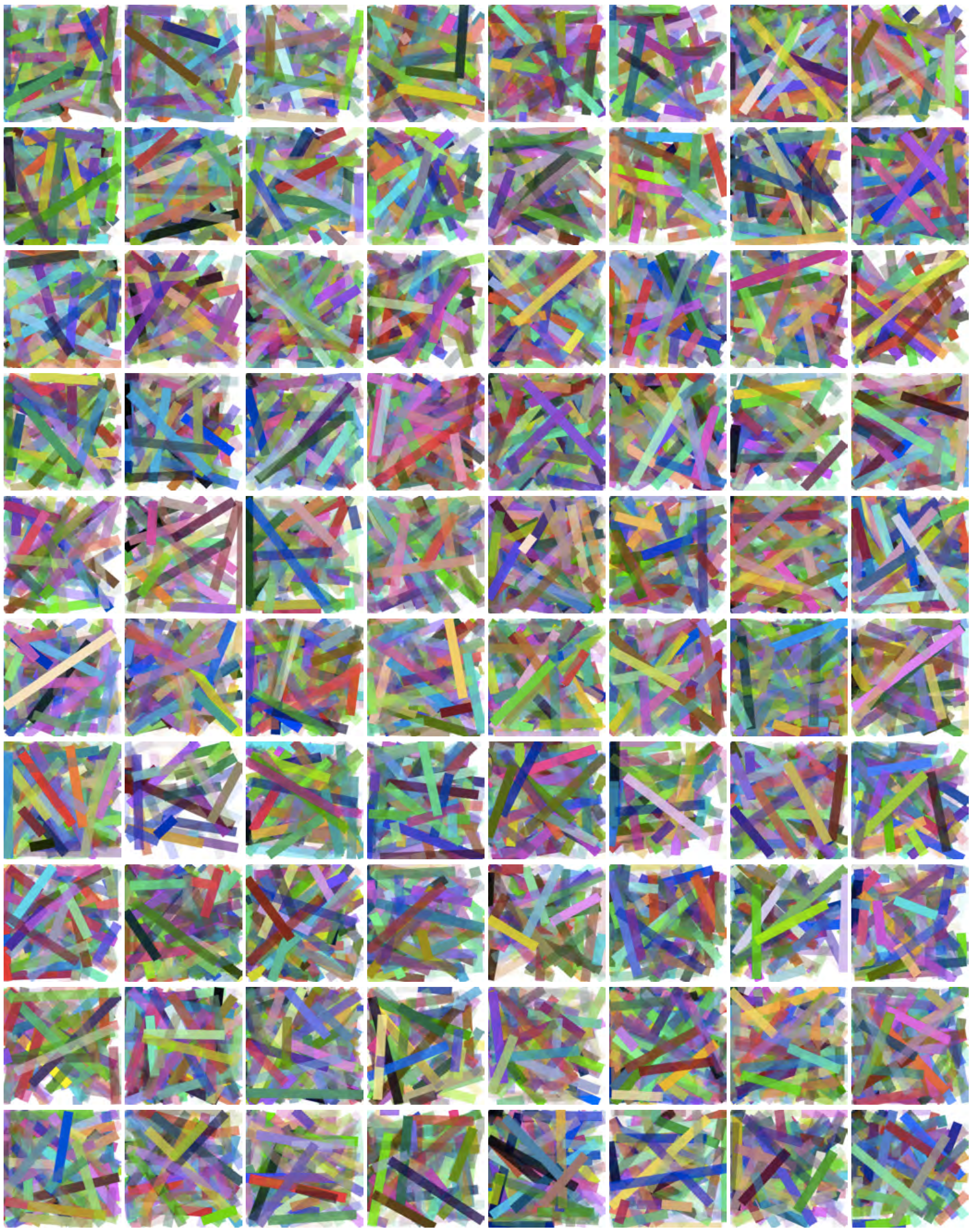

Figure 9 Lines visualization of public keys from author's PGP keyring. This is a key-only visualization.

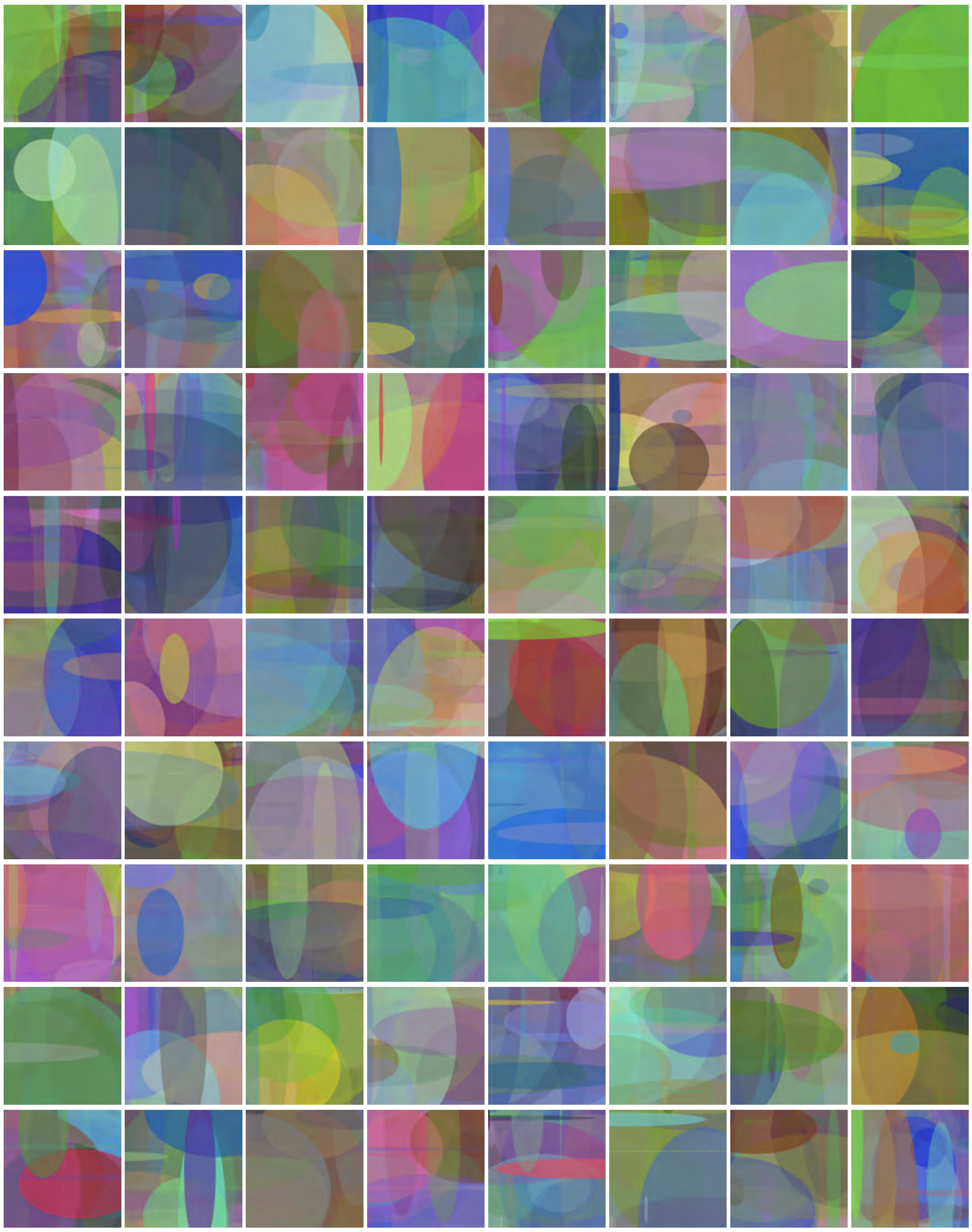

Figure 10 Ellipse visualizations of both key and a SHA-256 hash in a single image. The public keys originate from author's PGP keyring.

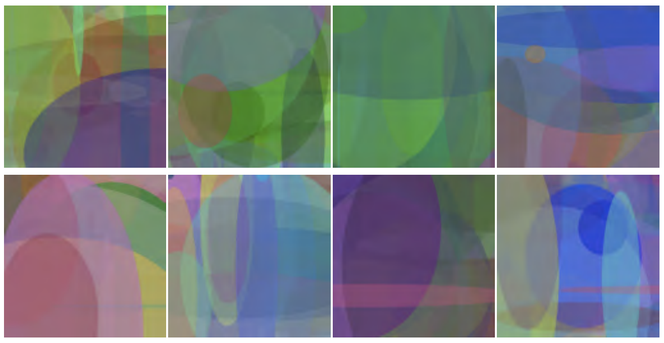

Figure 11 Ellipse visualization, keys only.

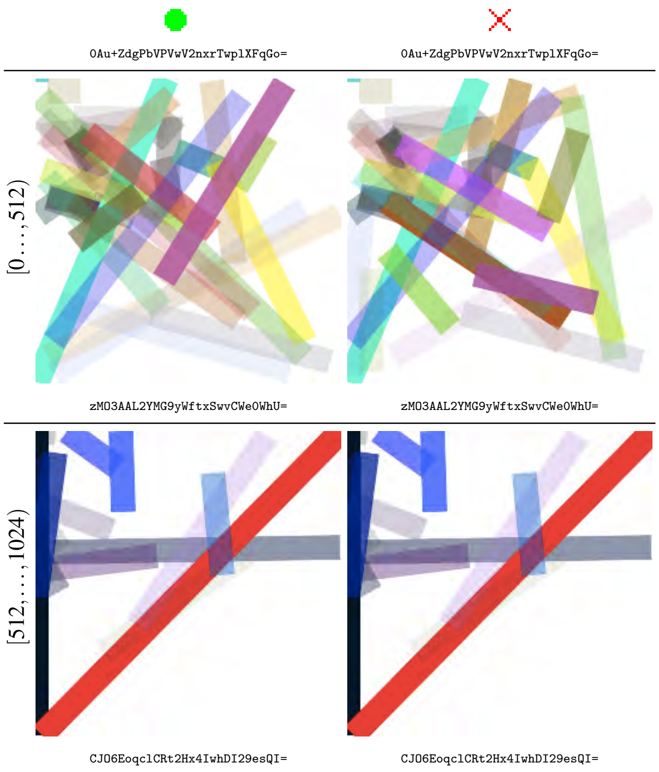
0Au+ZdgPbVPVwV2nxrTwplXFqGo=
0Au+ZdgPbVPVwV2nxrTwplXFqGo=
[0...,512)
zMO3AAL2YMG9yWftxSwvCWe0WhU=
zMO3AAL2YMG9yWftxSwvCWe0WhU=
[512,...,1024)
CJO6EoqclCRt2Hx4IwhDI29esQI=
CJO6EoqclCRt2Hx4IwhDI29esQI=

Figure 12 Detecting differences in real SHA1 collision data. We process 512 bytes at once as a single block, visualized together with SHA1 hash for the block. We also show the SHA1 hash for each block below its visualization. Left and right shows two different files, as top line shows. Their SHA1 hashes (shown in Base64) are, however, identical. The files are mostly identical, as visualized further 512 B block in the bottom row shows. The block $[512,\ldots,1024)$ features a lot of zeros and close to zero values, hence it looks more sparse in our visualization. The blocks in the middle row $[0,\ldots,512)$ are noticeably different in the visualization; this is clearly visible. However, their SHA1 hashes are the same. Thus, key spoofing based on hash collisions can be detected with our method.

tion from Open SSH [3] is used in production. It visualizes hashes as an ASCII-art. Displaying an image and not an ASCII-art should be a non-issue nowadays. However, it has the same weakness as other hash visualizations. All cryptographic security visualizations known to us, visualize hashes of the keys. This makes them prone to hash collisions. Our method does not have this weakness. We enhance security by visualizing the keys directly and optionally adding the hash visualization.

Even if the current implementation of our method omits the last few (< 8) bytes from the input, it might be possible to reconstruct or at least approximate parts of the key from the visualization. Hence, only public keys (which are available to anyone per design) should be visualized with our method.

5 Conclusions and future work

We present an approach to visualize cryptographic public keys with their hashes. Unlike previous approaches, our visualization is direct. Our method produces aesthetically pleasant visualizations (Fig. 10). Further, when both the key itself and its hash are visualized, our approach makes key spoofing and similar man-in-the-middle attacks much more complicated for the attacker. We made two binary data blocks with the same SHA1 hash visually distinguishable (Fig. 12).

One possible idea for the future work is to spend more bytes for further decisions during image generation. Possible options include using multiple visualization methods in one image and using gradients instead of solid colors. Line thickness could be also utilized as an input parameter, though for such visualizations (Fig. 4), a similar kind of transparency heuristics is needed as we introduced for our ellipse visualizations.

While visualizing binary data and its hash, we treat both as equal binary data. It makes sense to visualize data and hash in the same image, but with different methods, e. g. “ellipse” for data and “lines” for hash. Because presented method provides additional security, it would make sense to integrate it into cryptographic software and sensible communication protocols, e. g. online banking.

Acknowledgment

The author thanks Dr. Andreas Kokott for the discussion of possible online banking improvements with the presented visualization.

Supplementary material

Supplementary material showing more visualizations is available in the Zenodo repository under `https://dx.doi.org/10.5281/zenodo.817656`.

References

1. Federal information processing standards: Secure hash standard (SHS). Tech. Rep. FIPS PUB 180-4, Information Technology

Laboratory, National Institute of Standards and Technology (2015). https://dx.doi.org/10.6028/NIST.FIPS.180-4

2. Awni, J.: Cryptographic key visualization (2017). US Patent App. 14/837,652. Publication # US20170061199 A1
3. BSD General Commands Manual: Manual page for ssh – OpenSSH SSH client (2017)
4. Cervesato, I., Durgin, N.A., Lincoln, P.D., Mitchell, J.C., Scedrov, A.: A meta-notation for protocol analysis. In: Proceedings of the 12th IEEE Computer Security Foundations Workshop, pp. 55–69 (1999). https://dx.doi.org/10.1109/CSFW.1999.779762
5. Cheng, Y.M., Wang, C.M.: A high-capacity steganographic approach for 3D polygonal meshes. The Visual Computer **22**(9), 845–855 (2006). https://dx.doi.org/10.1007/s00371-006-0069-4
6. Cheng, Y.M., Wang, C.M.: An adaptive steganographic algorithm for 3D polygonal meshes. The Visual Computer **23**(9), 721–732 (2007). https://dx.doi.org/10.1007/s00371-007-0147-2
7. Conti, G., Grizzard, J., Ahamad, M., Owen, H.: Visual exploration of malicious network objects using semantic zoom, interactive encoding and dynamic queries. In: IEEE Workshop on Visualization for Computer Security, VizSEC '05, pp. 83–90 (2005). https://dx.doi.org/10.1109/VIZSEC.2005.1532069
8. Cox, I., Miller, M., Bloom, J., Fridrich, J., Kalker, T.: Digital watermarking and steganography. Morgan Kaufmann (2007)
9. Dhamija, R., Perrig, A.: Déjà vu: A user study. Using images for authentication. In: USENIX Security Symposium, vol. 9, pp. 4–4 (2000)
10. GNU Privacy Guard: Manual page for gpg2 – OpenPGP encryption and signing tool (2016)
11. Hou, Y.C.: Visual cryptography for color images. Pattern Recognition **36**(7), 1619–1629 (2003). https://dx.doi.org/10.1016/S0031-3203(02)00258-3
12. Liang, J., Lai, X.J.: Improved collision attack on hash function MD5. Journal of Computer Science and Technology **22**(1), 79–87 (2007). https://dx.doi.org/10.1007/s11390-007-9010-1
13. Naor, M., Shamir, A.: Visual cryptography, pp. 1–12. EUROCRYPT '94. Springer (1995). https://dx.doi.org/10.1007/BFb0053419
14. Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.S.: Malware images: Visualization and automatic classification. In: Proceedings of the 8th International Symposium on Visualization for Cyber Security, VizSec '11, pp. 4:1–4:7. ACM (2011). https://dx.doi.org/10.1145/2016904.2016908
15. Oliva, A., Torralba, A.: Modeling the shape of the scene: A holistic representation of the spatial envelope. International Journal of Computer Vision **42**(3), 145–175 (2001). https://dx.doi.org/10.1023/A:1011139631724
16. OpenSSL: Manual page for openssl – OpenSSL command line tool (2016)
17. Perrig, A., Song, D.: Hash visualization: A new technique to improve real-world security. In: International Workshop on Cryptographic Techniques and E-Commerce, CrypTEC '99, pp. 131–138 (1999)
18. Rescorla, E.: HTTP over TLS (2000). Request for Comments: 2818
19. Schneier, B.: Applied cryptography: protocols, algorithms, and source code in C. John Wiley & Sons (2007)
20. Stevens, M.: Counter-Cryptanalysis, pp. 129–146. CRYPTO '13. Springer (2013). https://dx.doi.org/10.1007/978-3-642-40041-4_8
21. Stevens, M., Bursztein, E., Karpman, P., Albertini, A., Markov, Y.: The first collision for full SHA-1. URL http://shattered.it/static/shattered.pdf
22. Subhedar, M.S., Mankar, V.H.: Current status and key issues in image steganography: A survey. Computer Science Review **13**, 95–113 (2014). https://dx.doi.org/10.1016/j.cosrev.2014.09.001
23. Suo, X., Zhu, Y., Owen, G.S.: Graphical passwords: a survey. In: 21st Annual Computer Security Applications Conference, ACSAC '05. IEEE (2005). https://dx.doi.org/10.1109/CSAC.2005.27
24. Teoh, S.T., Jankun-Kelly, T., Ma, K.L., Wu, S.F.: Visual data analysis for detecting flaws and intruders in computer network systems. IEEE/ACM Transactions on Networking **6**(5), 515–528 (1998)
25. Wang, X., Yin, Y.L., Yu, H.: Finding Collisions in the Full SHA-1, pp. 17–36. CRYPTO '05. Springer (2005). https://dx.doi.org/10.1007/11535218_2
26. Wang, X., Yu, H.: How to Break MD5 and Other Hash Functions, pp. 19–35. EUROCRYPT '05. Springer (2005). https://dx.doi.org/10.1007/11426639_2
27. Zimmermann, P.R.: The official PGP user's guide. MIT Press (1995)