Coded Hashes of Arbitrary Images

(or: the last frontier of emoji encoding)

Steven R. Loomis
srl@icu-project.org
(individual contribution)

Keith Winstein
keithw@cs.stanford.edu
Stanford University

2016-05-02

https://srl295.github.io

1 Introduction

> Emoji are pictographs (pictorial symbols) that are typically presented in a colorful cartoon form and used inline in text. [...] In Unicode 8.0, there is a total of 1,282 emoji, which are represented using 1,051 code points.[1]

Recently, there has been considerable interest in adding newly created pictorial symbols, not found in any existing character set, to the Unicode Standard as emoji.[2] Advocacy groups and others request these code points because Unicode plain text remains the dominant interoperable interchange format for messaging. In practice, before a new emoji can be used, a code point must be assigned and be recognized by the sender and receiver. However, a longer-term goal for Unicode is that implementations should support "embedded graphics, in addition to the emoji characters".[3]

In this proposal, we describe a mechanism to uniquely identify arbitrary images within a plain-text Unicode character sequence. This will allow implementers to create their own emoji without needing to request and wait for the assignment of a code point. The basic idea is to encode a globally-unique *secure hash* of the emoji in a Unicode character sequence. Once the receiver knows the image's hash, it may already have the corresponding image, or may have a choice of several mechanisms to retrieve it.

Our technique will gracefully degrade on legacy Unicode implementations, but our proposal is limited to allowing Unicode to *uniquely identify* an arbitrary image. We propose to leave to implementers the details of how to retrieve the actual image.

[1] Mark Davis and Peter Edberg. *Unicode Technical Report 51: Unicode Emoji*. 2015. URL: http://www.unicode.org/reports/tr51/.

[2] *E.g.,* Taco Emoji campaign, Beard Emoji campaign, Dumpling Emoji campaign. There are currently 79 candidate emoji that have been assigned tentative code points.

[3] Davis and Edberg, *Unicode Technical Report 51: Unicode Emoji*, op. cit., Section 8, "Longer Term Solutions".

2 Proposal

Outline: implementations will be able to create "implementation-defined emoji," and allow their users to send them to receivers. The sender encodes a secure hash of the emoji (which serves as an unforgeable globally unique identifier) in a sequence of coded characters, which we call a Coded Hash of an Arbitrary Image (CHAI). Upon receiving the CHAI sequence, the receiving implementation may already know the corresponding image, or may have to request it either from the sender or from a third party.

2.1 Representing the emoji

The first step to creating an implementation-defined emoji will be to represent the emoji in a canonical form. This format will need to be specified, but we do not do so here. As a strawman, we suggest a JSON document with three keys: `content-type` (any IANA-registered MIME Media Type), `image` (a Base64-encoded suggested rendering of the emoji, interpreted as the format named in the `content-type` field), and `name` (name of the emoji, which may be used in fallback rendering for visually-impaired users). We call this representation the "emoji description."

2.2 Secure hash

The implementation will identify the emoji by taking the SHA-256 hash of the emoji description. Given a secure hash value, it is intended to be intractable to find a different input that produces the same hash value. This is known as "second-preimage resistance"; SHA-256 is believed to have strong second-preimage resistance.

2.3 Code points allocated to express the secure hash

We propose to allocate 65,536 (2^{16}) code points to represent bits of the secure hash. Each code point will express 16 bits of the secure hash. We envision allocating Plane 13, from U+D0000 through U+DFFFD. (The two code points at the end of the plane are noncharacters.) These 65,534 code points will represent the 16-bit values 0000_{16} through $FFFD_{16}$. In addition, we envision allocating U+EFFFC to represent the value $FFFE_{16}$, and U+EFFFD to represent the value $FFFF_{16}$. We refer to these 65,536 code points as "CHAI characters." Each code point will have general category `Cf`.

In UTF-8, UTF-16, and UTF-32, every code point outside the BMP requires 32 bits to encode. As a result, the efficiency of this scheme will be 50% in each Unicode encoding scheme (16 bits of the hash will consume 32 bits in memory, on disk, or on the wire).

2.4 Encoding an implementation-defined emoji

To code a CHAI, the sender first encodes a "fallback" base character that most closely approximates the implementation-defined emoji. This will allow rendering to degrade gracefully

on receivers that do not support CHAIs.

Next, the sender encodes the secure hash by appending between between one and sixteen CHAI characters. This will represent a prefix of the SHA-256 hash of the emoji description. The sender chooses how many bits of the hash to include. (We expect 80 bits, or five CHAI characters, to be sufficient for typical use today. The risk of including too few bits is that the hash may no longer be globally unique, especially in the presence of an attacker who wishes to create a different image with the same hash prefix.)

Therefore, the actual encoding is:

`<basechar> + <CHAI char> + <CHAI char> + <CHAI char> …`

where the *basechar* represents a fallback base character, and each CHAI character (from the range U+D0000 .. U+DFFFD and U+EFFFC .. U+EFFFD) represents 16 bits of the secure hash of the emoji description.

2.5 Receiving and rendering a CHAI

On receipt, the receiver determines if it already knows of an emoji whose hash matches the prefix encoded in the CHAI characters. If so, it displays the emoji (either using the image provided in the emoji description, or an image known locally). Otherwise, the receiver displays the base character while it attempts to retrieve an emoji description whose hash matches the encoded hash prefix.

We do not specify how this retrieval will take place; we expect it to vary based on the messaging protocol. In some protocols, it might be easiest for the receiver to simply ask the sender to send the full emoji description. In others, the receiver might consult an online repository run by the implementing vendor. Alternately, the receiver might consult a community repository where anybody can contribute any emoji description.

Because the CHAI serves as an unforgeable globally unique identifier for the emoji description, the receiver is intended to have confidence that if it finds a matching emoji description from anywhere, it can display it per the sender's intent.

Depending on the protocol, as the input arrives, the receiver may have some ambiguity about when the sequence of CHAI characters ends. A receiver may choose to wait until the next non-combining character (signaling the end of the combining character sequence), or a protocol-defined end-of-message signal, before retrieving the emoji description.

2.6 Privacy

As Unicode Technical Report #51 notes: "There are also privacy aspects to implementations of embedded graphics: if the graphic itself is not packaged with the text, but instead is just a reference to an image on a server, then that server could track usage."

We agree. A similar exposure could occur if the protocol provides for the receiver to ask the sender to send an unknown emoji description; this could unwittingly expose if anybody *else* has ever sent the same emoji to the receiver.

It is not the goal of this document to address all aspects of embedded graphics, and we intend to leave to domain experts and implementers the decision on how to handle these

questions. For example, a client might behave similarly to current practice in retrieving external images in email. If the receiver already knows of the identified emoji, it can display it, but otherwise the receiver would display the fallback (base) character and ask the user to push a button before retrieving the emoji description.

3 Sample Code

Sample code here is for use with Node.js version 6.0.0

3.0.1 chai-encode.js

```
// chai-encode.js
const fs = require('fs');
const crypto = require('crypto');

// base char
const base = '\u2615';

// emoji description
const description = {
    name: 'CHAI',
    "content-type": 'image/png',
    image: fs.readFileSync('chai.png').toString('base64')
};

// Write out canonical emoji description
const description_json = JSON.stringify(description);

console.log('# Writing','chai.json');
fs.writeFileSync('chai.json', description_json);

// Now, hash it
const hash = crypto.createHash('sha256')
                   .update(description_json)
                   .digest();
const hashHex = hash.toString('hex');
console.log('# SHA-256 hash:', hashHex);

const hashChars = 5;
console.log('# Going to use', hashChars,
     'CHAI chars or', (hashChars*16), 'bits.');

var outString = base;
```

```
console.log('# <U+'+base.codePointAt(0)
                        .toString(16)
                        .toUpperCase()+'> (base)');
for (var i = 0; i<hashChars; i++) {
    // get the next 16 bits
    var hashBits = (hash[(i*2)+0] << 8) | (hash[(i*2)+1]);
    // console.log('Chunk', hashBits.toString(16));
    var outPoint;
    if( hashBits >= 0xFFFE) {
        // FFFE -> EFFFC, FFFF -> EFFFD
        outPoint = (0xE0000 | hashBits) - 2;
    } else {
        outPoint = (0xD0000 | hashBits);
    }
    console.log('# <U+'+outPoint.toString(16).toUpperCase()+'>',
        'IMAGE HASH', hashBits.toString(16).toUpperCase());
    outString = outString + String.fromCodePoint(outPoint);
}

console.log('# Writing chai.txt');
fs.writeFileSync('chai.txt', outString);
```

Sample run

```
# Writing chai.json
# SHA-256 hash: 37d8c5403d29ec7d6f59b02690414de77b0597735a953c2a00fbea50b5f79bb5
# Going to use 5 CHAI chars or 80 bits.
# <U+2615> (base)
# <U+D37D8> IMAGE HASH 37D8
# <U+DC540> IMAGE HASH C540
# <U+D3D29> IMAGE HASH 3D29
# <U+DEC7D> IMAGE HASH EC7D
# <U+D6F59> IMAGE HASH 6F59
# Writing chai.txt
```

3.1 chai.json

Example Canonical Emoji Description (output of `chai-encode.js`)


```
{"name":"CHAI","content-type":"image/png","image":"iVBORw0KGgoAAAANSUhEUgAAAEAAAABACAYAAACqaXHeAAACT0lEQVR42u2ZMUvDQBiGS4YM4tChU/
      EHiFMRp45OpTg4dBR/gENnp0IH8QdIR2lTTdfqoOBQRRAEG8VNOnVy6iTFsX7epReJIU2rufQu8D7wUS5pL+/39
      vLl7pLJAAAAAAAAAAAAYXyyyKZEa1bolQqxqKbEgKrQK92AQUoMGCRhwGMSnSYECb1S2UuZAXuyOltjcSM65ZHXPPm80GnK6KzmS9wzoa65AXVZI7UWkjBvjzQ3YCTDgNqMf/
      sqBXWAhE6pyRfEsXFKDNiRlXzBVwN4bPJPcizZMWbxzuKNntt96ls9clpdctoW9VsN9nnM2ofs3AFr75NztksvrW16am6R01ynVytPD6erpmluCJ1GHPeOAm0SRvwcMwzjhF/
      QvTAXwIVwQRedD7q2u65QLpgLnyZguQnxxHiCPNFpwmOZRn71WxR3hPIflwPtsO9MQkXcnhFddhYT3O38juB53k/v/
      E8GcF0yDGjMMcBO6DaINnMBc4VeO64BwzkGFJdiwD9GhNBbjGPAMJA0RRhVXLoJ0ckXZTyhGiEGDEOqqjvUNDPAlmFAOdBJyfckKPmOT5TcBvOH/
      ySuASuiozvfMcN3a3hrgqamBpzImElVfMnmZqwKvRmhoUnyhtCzLms6mfOZUImacmpiwE5SU/T7wFTYK4pesbzRxIBEF2krojg2ArWAdKkDqpbpdc0MWPpGzc/Wk+
      LkTZVbde7mo2IDlG7WutvPig1Qul0/UF0HVL+wqWpigLJXdtnM9CUkKY5sBgAAAAAAAAAAAAAAAP7ANw53SLcSCEY2AAAAAElFTkSuQmCC"}
```

3.2 chai.png

Sample PNG

3.3 chai-decode.js

```
//chai-decode.js
const fs = require('fs');
console.log('# Reading', 'chai.txt');
var str = fs.readFileSync('chai.txt').toString();
var hash = "";
function isCHAI(x) {
    return ( ((x>=0xD0000) && (x<=0xDFFFD))
            || (x===0xEFFFC) || (x===0xEFFFD) );
}
function add(ch) {
    // If the end of a CHAI sequence, emit hash.
    if(!isCHAI(ch)) {
        if(hash !== "") console.log('# CHAI Hash: ' + hash);
        hash = "";
        return;
    }
    // Else, accumulate
    // Fixup our two exceptions
    if (ch === 0xEFFFC) ch = 0xDFFFE;
    if (ch === 0xEFFFD) ch = 0xDFFFF;
    var hashBits = ch & 0xFFFF;
    console.log(' hash bits:',hashBits.toString(16).toUpperCase());
    // accumulate as a string
    hash = hash + hashBits.toString(16).toUpperCase();
}
for(var i = 0; i < str.length; i++) {
    var ch = str.codePointAt(i);
    if(ch > 0xFFFF) {
        i++;
    }
    console.log('U+'+ch.toString(16).toUpperCase());
    add(ch);
}
add(0); // print out trailing hash
```

Sample run

```
# Reading chai.txt
U+2615
U+D37D8
 hash bits: 37D8
U+DC540
 hash bits: C540
U+D3D29
 hash bits: 3D29
U+DEC7D
 hash bits: EC7D
U+D6F59
 hash bits: 6F59
# CHAI Hash: 37D8C5403D29EC7D6F59
```

The resulting hash matches the first 80 bits of
`37d8c5403d29ec7d6f59b02690414de77b0597735a953c2a00fbea50b5f79bb5`

4 Character Data

65,536 characters are to be encoded.

4.1 Character Properties

```
D0000;IMAGE HASH 0000;Cf;0;BN;;;;;N;;;;;
...
DFFFD;IMAGE HASH FFFD;Cf;0;BN;;;;;N;;;;;
EFFFC;IMAGE HASH FFFE;Cf;0;BN;;;;;N;;;;;
EFFFD;IMAGE HASH FFFF;Cf;0;BN;;;;;N;;;;;
```

4.2 Line Breaking

```
D0000..DFFFD;CM   # Cf[65,534] IMAGE HASH
EFFFC..DFFFD;CM   # Cf     [2] IMAGE HASH
```

References

Davis, Mark and Peter Edberg. *Unicode Technical Report 51: Unicode Emoji*. 2015. URL: http://www.unicode.org/reports/tr51/.

Colophon

Typeset by LaTeX. Made with 100% recycled bits. All opinions belong to the authors and do not reflect the opinions of their associated employers.

Fork me on GitHub: `https://github.com/srl295/srl-unicode-proposals`