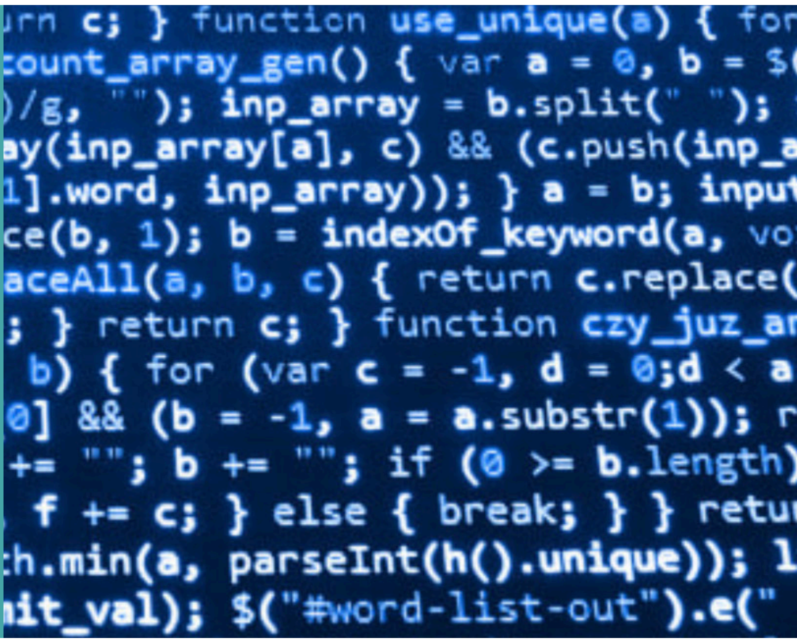
CCL SOLUTIONS GROUP | incorporating evidencetalks

IndexedDB on Chromium



With Chrome today's most popular browser and exponential growth in "Chrome-esque" applications, a deep understanding of how they store data is essential. CCL's Principal Analyst Alex Caithness is back with a follow-up deep dive look at IndexedDB - and an open-source gift to his fellow practitioners and researchers.

In my previous blog I laid out why I think that LevelDB is an important data format for those working in the broad field of digital forensics to consider. I discussed how LevelDB arranges its data on disk and how it relates to the web data storage API called IndexedDB. If you haven't read that blog, I would highly recommend taking a peep before going any further here, as a lot of this blog will refer back to the material there.

In this paper we're going to take a deep dive into the way that Chrome-esque applications manage their IndexedDB stored data: how LevelDB is used alongside the file system; how metadata is organised and structured; the encoding of keys for records; and finally how the serialised objects are stored. At the end of the blog you'll also find a link to a GitHub repository containing all of the Python Scripts we've generated so far during this research; we hope these will assist other people in the community to take this research forward and build upon this work.

IndexedDB – a quick re-primer

IndexedDB is a key-value store where the keys and values are JavaScript objects. IndexedDB is organised as databases per domain, which can each contain a number of object stores (analogous with tables in a classic relational database), the object stores then contain a number of JavaScript objects which can be addressed either via a primary key value or another indexed field inside the object.

In Chrome-esque applications, IndexedDB is (mostly – we'll get to this) built on top of a LevelDB store. LevelDB is a pure key-value store with raw binary keys and values. In order to understand how IndexedDB is being implemented, we need to know how the structure, metadata, and records of the IndexedDB store are translated into the raw keys and values in LevelDB.

The LevelDB store for a particular domain will be found in the IndexedDB folder with a name in the form: "[host with concurrent separators replaced an underscore]_[dbid].indexeddb.leveldb", e.g. for `"https://archive.org"` the folder would be named: `"https_archive.org_0.indexeddb.leveldb"`.

Primitive Data Encoding

Within keys and values the usual encoding format for integers and floating-point numbers, IEEE 754 double precision, will be encountered; integers should be stored in little-endian byte order, floating point numbers will use the host system's endianness (little-endian more often than not on desktops). Text is variously encoded using ASCII (although it's safer to assume these values are a raw blob as they can contain 8-bit data), UTF-8 and UTF-16. In the case of UTF-16, the big-endian variant is always used regardless of the host's endianness – this means that on most desktop systems you'll be switching endianness for text.

Varints

In addition to the usual suspects, many integer values are stored using a variable length integer format (henceforth "varint"). Varints, which are found in a number of Google data formats (as well as SQLite), are integer representations which expand to occupy as many bytes as are needed to represent their value. For example: the number "54" is small and fits in a single byte, whereas "1032" is larger and will be encoded across 2 bytes.

In the IndexedDB data, the varints are stored little-endian, and can be read by inspecting the most significant bit of each byte; if that bit is 1, then the following byte should also be read; repeat until the most significant bit is not set, or until all bytes have been consumed for the maximum size of the varint (10 bytes for a 64-bit varint, 5 for a 32-bit varint). You may then discard the most significant bit from each byte and arrange the remaining bits in little-endian order to get the value.

The pseudo-code below shows the algorithm (take "buffer" to be an array of bytes containing the prospective varint):

```
result = 0
idx = 0
limit = 5 if is_32_bit else 10
while idx < limit:
  result |= ((buffer[idx] & 0x7f) << (idx * 7))
  idx += 1
  if buffer[idx] & 0x80 == 0:
      break
return result
```

One downside of varints is that to represent negative numbers using a traditional two's complement scheme requires the varint to occupy its maximum number of bytes so that the most significant bit can be set to 1 – this makes varints very wasteful when negative numbers need to be stored often. To balance this, Google have opted for a "ZigZag" encoding when signed numbers need to be stored using varints. In a Signed ZigZag varint, the number is stored as an unsigned (positive) integer and then odd numbers are taken to be negative e.g.:

Raw unsigned Value	ZigZag value
0	0
1	-1
2	1
3	-2
4	2
5	-3

Taking "raw" to be the original unsigned value, in pseudo-code this could be written as:

```
is_negative = raw & 0x01 > 0
result = raw >> 1
if is_negative:
  result = -result
return result
```

BigIntegers

JavaScript can store arbitrarily large (or, at least very, very big) integer values. The BigInt datatype in IndexedDB can be decoded thus:

Read a varint; the least significant bit gives you the sign of the number (1 for negative). Bit-shift the value left 4 bits; this shifted value tells you how many unsigned integers to read – 32-bit integers on 32-bit systems, 64-bit bit integers on 64-bit systems. Take the concatenated value and read it with the endianness of the system.

The Database Structure

In order to read through the objects in the IndexedDB LevelDB database in a meaningful way we need to understand the structure of the IndexedDB: how many object stores there are; how to identify them; and so on. We can achieve this by identifying records in the LevelDB backing store with specially constructed metadata keys.

The following understanding is pulled from the technical documentation and code found in the following links:

- https://github.com/chromium/chromium/blob/master/content/browser/indexed_db/docs/leveldb_coding_scheme.md
- https://github.com/chromium/chromium/blob/master/content/browser/indexed_db/indexed_db_leveldb_coding.h
- https://github.com/chromium/chromium/blob/master/content/browser/indexed_db/indexed_db_leveldb_coding.cc

As we proceed, please keep at the back of your mind the fact that (as I discussed in the previous blog) it is trivial in a great many cases to recover deleted data from LevelDB databases; indeed in many cases it's more work to ensure that you're looking at the current live data. It is quite possible for us to encounter multiple LevelDB records with identical keys if we are reading raw LevelDB records; only if the record has a live state and has the highest sequence number for that key, is that record the current version of a live record. It is, of course, quite beneficial in a number of situations to be able to consider deleted records, but I can't think of a great many uses for that when considering the IndexedDB metadata, save for detecting that an entire object store has been programmatically deleted. As a result, in the following sections where we're looking at the metadata structures, please assume that I'm only talking about the current live version of each record.

Key Prefix Structure

The keys of the records relating to IndexedDB data and metadata all have a common prefix format which indicates what aspect of IndexedDB each record refers to. Strictly speaking, this prefix is of a variable length, but unless there are more than 255 IndexedDB databases, more than 255 object stores in a single database, or more than 255 indexes for a particular object store, the key prefix will be 4 bytes long.

The key prefix contains 3 integer values (always stored with a little-endian byte order): the database ID, the object store ID, and the index ID. For data records, the IDs are enumerated from 1, so if any value is 0 then the key is either referring to metadata or to data which is not specific to a particular database, object store or index. Furthermore, the index IDs 1-3 inclusive are reserved with a special meaning (more on that later).

Every key prefix begins with a byte which gives the size in bytes for the three values which follow directly after. The sizes are stored in three bit fields within this byte: the top 3 bits contain one less than the size of the database ID; the next three bits contain one less than the size of the object store ID; the bottom two bits contain one less than the size of the index ID.

```
0bxxxyyyzz

xxx: database ID size - 1

yyy: object store ID size - 1

zz:  index ID size - 1
```

In the current implementation, unless there are more than 255 databases, or more than 255 object stores or indices in any of those databases, then those values will also fit in a single byte. As the bit fields store one less than the size in bytes, that will be a 0 for all 3 sizes; so, in a great many cases the key prefix will start with 0x00 and the key prefix will be 4 bytes in length.

StringWithLength

Most text fields in the database metadata will be stored as "StringWithLength" structures. This structure comprises a varint value giving the length of the string in characters, followed by the string encoded using UTF-16; as UTF-16 is made up of two-byte units, you need to multiply the length value by 2 to get the length in bytes.

Global Metadata

Global metadata does not pertain to any particular database, object store or index, so in the key prefix those values will all be zero, therefore a global metadata record can be identified by the key prefix: `0x 00 00 00 00`.

The next byte will define the type of metadata being defined, the structure of the remainder of the key (if any) and the structure of the record's value.

For our purposes, there are two types that we are most interested in:

- `1 (0x01)` – there will be no further data in the key and the value of the record will contain a varint which gives the highest IndexedDB database ID.
- `201 (0xC9)` – there will be as many "global metadata 0xC9 keys" as there are IndexedDB databases represented in the LevelDB store. The key will continue with the origin (domain) for the database stored as a StringWithLength followed by the database name, also as a StringWithLength. The value of the record will contain the database ID for the database as a varint.

These two values are important as they will inform how to read other metadata fields which refer to particular databases.

Database Metadata

For database metadata records, the key prefix will contain the database ID for the database in question, then zero for both the object store ID and the index ID. The key prefix for these records will therefore be: `0x 00 yy 00 00`, where yy is the database ID.

The next byte will define the type of metadata being defined. Of particular interest to us:

- `0 (0x00)` – the origin (domain): this value was not always populated in my test data, but it can be read from the global metadata 0xC9 records instead.
- `1 (0x01)` – the database name: this value was not always populated in my test data, but it can be read from the global metadata 0xC9 records instead.
- `3 (0x03)` – maximum allocated object store ID: the value is a varint which gives the highest (inclusive) object store in the IndexedDB database. If the database holds any object stores at all, this will be at least 1. This value is required to read metadata about, and records from object stores.

Object Store Metadata

The key prefix format for object store metadata is a little surprising – as with the database metadata only the database ID field is set, with the object store and index ID both still set to zero (this is because when the object ID is set to a value, the records contain record data or metadata referring to records rather than the object store itself). Instead the standard key prefix structure will be followed by the value `50 (0x32)`, e.g. for database ID 1: `0x 00 01 00 00 32.`

This prefix will then be followed by the object store ID encoded as a varint; for object store ID 1 this would be: `0x 00 01 00 00 32 01`. The key is then completed by a byte which determines the type of metadata to follow; of note:

- `0 (0x00)` – object store name: the value will contain the name of the object store encoded using UTF-16.

Reading Records

- Once we understand the "shape" of the IndexedDB database, we can start to consider how to identify and read the actual records in the object stores.

Keys

The LevelDB records relating to data held in an IndexedDB object store can be identified by key prefixes where the database ID, object store ID and index ID are all populated. Index IDs 1 through 3 are all reserved and have a special meaning:

- 1: A record in the object store identified by its primary key
- 2: An "exists" marker with the version of the record as its value
- 3: An external object table record which details file or blob data related to the record that isn't stored with the record itself

The standard key prefix format in all cases is followed by the "user key" – for records with index IDs 1-3 this will be the JavaScript object used as the primary key for the object stored in IndexedDB; for all other index IDs, this will be a JavaScript object which is being used as an index.

The user key is encoded using the IdbKey structure which is a tagged format – a type byte followed by the corresponding data:

Byte	Type	Encoding
0x00	Null	No further data, null is an empty key - strictly speaking should not appear in real data
0x01	String	StringWithLength (varint length in characters, followed by UTF-16 BE string)
0x02	Date	A double precision floating point number - milliseconds elapsed since mid-night 1970-01-01
0x03	Number	A double precision floating point number
0x04	Array	A varint giving the number of elements in the array, followed by that many IdbKey structures
0x05	MinKey	Currently unknown - but code shows that it has no further data and the docu-mentation doesn't list it as something that should occur in a key
0x06	Binary	Varint length, followed by that many bytes

The code responsible for reading and writing IdbKey structures can be found in (the function "`DecodeIDBKeyRecursive`" is probably the easiest way to read the algorithm):

- https://github.com/chromium/chromium/blob/master/content/browser/indexed_db/indexed_db_leveldb_coding.cc

For the reserved index IDs (1-3), the IdbKey will complete the key data; for other records the first IdbKey structure will be followed by a varint sequence number, and the primary key for the record (again encoded as an IdbKey structure).

Record Data

If we are only interested in reading stored objects from IndexedDB, then the only index ID that we really need is 1 (the record identified by its primary key); ID 3 will also be of interest in most cases, but we'll get to that later. The value for records following this key structure will contain the serialised form of the object stored in IndexedDB.

The method by which the JavaScript objects are stored is governed by two different areas of the Chromium application code: primarily the V8 JavaScript engine (https://v8.dev/), with additional work done by the Blink rendering engine (https://www.chromium.org/blink).

The relevant code from V8 can be found here:

- https://github.com/v8/v8/blob/master/src/objects/value-serializer.cc

In Blink, most of the work is done across these three source files:

- https://chromium.googlesource.com/chromium/src/third_party/+/master/blink/renderer/bindings/core/v8/serialization/serialization_tag.h
- https://chromium.googlesource.com/chromium/src/third_party/+/master/blink/renderer/bindings/core/v8/serialization/v8_script_value_deserializer.h
- https://chromium.googlesource.com/chromium/src/third_party/+/master/blink/renderer/bindings/core/v8/serialization/v8_script_value_deserializer.cc

The links above reference the master branch of the repositories, so if anything changes in the format, they should always point you to the most up-to-date version. If you require a view of the code from the time that the blog was written, a tag from the star of October 2020 should be used instead.

I will do my very best to summarise how the format operates, but if you are in any doubt then the source code can provide definitive answers (albeit in C++).

Backing Store Version

The record data starts with a varint giving the version of the record.

Blink Version Tag

The record data continues with a version tag, which gives the version of the Blink rendering engine that was involved in the serialisation of the data. The version tag takes the form of a 0xff byte followed by a varint giving the version.

V8 Version Tag

The V8 version tag works in the same manner to the Blink version tag – a `0xff` byte followed by a varint.

Object Encoding

After the 3 values that form a header of sorts, the rest of the data is made up of the serialised JavaScript object. The serialisation uses a tagged format: a tag byte followed by the serialised data.

The datatypes can be split into two broad camps: primitive values and objects. The primitive values can be read without any side effects – they include the basic numeric types, Boolean types, basic strings and the like.

On the other hand, objects, once serialised, can be referred to by a reference ID if it is reused in the data (e.g. the same Date object which appears multiple times). The reference IDs for objects start from 0; every time an object is read, it can be added to a list, so it may be referenced subsequently by its index in that list. For most primitive value types there is also a "boxed object" version, which will additionally be added to the list of references. The object's reference ID should be assigned when the tag is first read, so for collection types, the ID should be assigned before the ID of any child objects.

The objects may be further grouped into basic objects and property objects – we'll get to the property objects later as they warrant additional explanation. The tables below describe the encoding of primitive values and basic objects:

Primitive Values		
Tag	**Primitive Type**	**Format**
_ (0x5f)	Undefined	No further data
- (0x2d)	"The Hole"	No further data - used to represent gaps in arrays etc.
0 (0x30)	Null	No further data
T (0x54)	True	No further data
F (0x46)	False	No further data
(0x00)	Padding	Ignore all instances of this tag
U (0x55)	Uint32	Stored as a varint (with a 32-bit limit)
I (0x49)	Int32	Stored as a vZigZag arint (with a 32-bit limit)
N (0x4e)	Double	Double precision floating point number
Z (0x5a)	BigInt	Stored using BigInt encoding
S (0x53)	UTF-8 String	Varint length followed by that many bytes, interpret as UTF-8
" (0x22)	1 Byte String	Varint length followed by that many bytes, may be ASCII, safer to treat as a byte array
c (0x63)	2 Byte String	Varint length followed by that many bytes, interpret as UTF-16 BE

A special case is the tag "V" (0x56) which is only allowed to be encoded directly after an array buffer (or a reference to an array buffer). This represents an "array buffer view" which interprets the data found in the preceding array buffer as a sequence of numerical values. The array buffer view does not get assigned an object ID or get appended to the list of previous objects.

The data following the "V" tag takes the format of 3 varints (although the first one should always fall within an 8-bit range) which give the type of the view followed by the offset and length of the data in the array buffer to be interpreted. The type tags for the array buffer view are as follows:

Tag	Type
b (0x62)	8-bit signed integer
B (0x42)	8-bit unsigned integer
C (0x43)	8-bit unsigned integer (clamped)
w (0x77)	16-bit signed integer
W (0x57)	16-bit unsigned integer
d (0x64)	32-bit signed integer
D (0x44)	32-bit unsigned integer
f (0x66)	32-bit floating point number
F (0x46)	64-bit floating point number
q (0x71)	64-bit signed integer
Q (0x51)	64-bit unsigned integer
? (0x3f)	"Data view" (a slice of the underlying buffer)

Property objects contain collections of objects and/or primitive values. In most cases, these collections are stored as a sequence of keys and values (arrays in JavaScript are essentially a key-value collection with numerical indexes as keys). The exception in this group is the "Set" type which is a collection of unique objects/values, but as it shares structural similarities with the other data types in this group, I'll consider it at the same time.

In addition to the tag identifying their type (the start tag), during the reading of these object types there will be an operation which involves reading a number of child items; these child items are no different to any value or object in the serialised data – using the same tagged format and sharing the same rules on IDs and referencing. The number of child items is not pre-determined, rather the reader is expected to read objects until the "end tag" for the collection is found.

Property Objects		
Start tag	**End tag**	**Type**
o (0x6f)	{ (0x7b)	Object (keys and values)
a (0x61)	@ (0x40)	Sparse array
A (0x41)	$ (0x24)	Dense array
' (0x27)	, (0x2C)	Set
; (0x3b)	: (0x3a)	Map

The algorithm for reading each type is described below:

Object:

- Until the end tag is encountered, keep reading key and value objects from the data (the key is written first, followed directly by the value, the end tag should not interrupt a key-value pair).
- Read a varint – this should be the total number of keys in the object.

Sparse Array:

- Read a varint – this is the total size of the array.
- Until the end tag is encountered, keep reading key and value objects from the data (the key is written first, followed directly by the value, the end tag should not interrupt a key-value pair). The key should be an integer (or a value that will convert to an integer), which is the index in the array for the value.
- Read a varint – this should be the number of keys read.
- Read a varint – this is the total size of the array, restated.

Dense Array:

- Read a varint – this is the total size of the array.
- Read that many objects – these will be the items in the array in order.
- Until the end tag is encountered, keep reading key and value objects from the data (the key is written first, followed directly by the value, the end tag should not interrupt a key-value pair). The key should be an integer (or a value that will convert to an integer), which is the index in the array for the value. I believe from comments in the code, that the sparse section of the array is to allow for just-in-time updates to the array in case it has changed during writing – it's usually empty.
- Read a varint – this should be the number of keys read.
- Read a varint – this is the total size of the array restated.

Set:

- Until the end tag is encountered, keep reading objects – these are the elements in the set.
- Read a varint – this should be the number of objects read.

Map

- Until the end tag is encountered, keep reading key and value objects from the data (the key is written first, followed directly by the value, the end tag should not interrupt a key-value pair).
- Read a varint – this should be the total number of keys in the map.

There are a number of other tags which are supported by the V8 serialisation format, but as far as I can tell they aren't used in IndexedDB; or at least I have so far not been able to generate a test case where they are:

Tag	Type
u (0x75)	Shared Array Buffer
t (0x74)	Array Buffer Transfer
r (0x72)	Error
w (0x77)	WASM Module Transfer
m (0x6d)	WASM Memory Transfer

The final tag in the V8 serialisation format that we need to discuss is "\" (0x5c). This represents the start of a "host object", i.e. an object for which the serialization is delegated to the host application, which in this case is Chromium's Blink rendering engine.

Theoretically, the format here could be anything that the host application designates, but luckily Blink uses a format which shares many of the same principles seen in the "outer" V8 format, i.e. it is primarily a tag-based format. The full list of tags are found in: https://chromium.googlesource.com/chromium/src/third_party/+/master/blink/renderer/bindings/core/v8/serialization/serialization_tag.h, but as it doesn't seem like all of them are used in IndexedDB, I want to focus on the two which I was readily able to generate and that I have seen represented widely in real world databases – these relate to the storage of files within IndexedDB.

Within the code there are two different ways that files can be represented within the Blink serialization: inline (within the serialized data) using the FileList and File tags ("l" / `0x6c` and "f" / `0x66`); or referencing an external store with the FileListIndex and FileIndex tags ("L" / `0x4c` and "e" / `0x65`). During testing, I was only able to force IndexedDB to use the latter reference types, even with very small files, so I suspect that is the default behaviour.

The encoding of these two types is very simple:

Tag	Type	Format
e (0x65)	FileIndex	Varint - file index number
L (0x4c)	FileListIndex	Varint count, followed by that many varint file index numbers

Not a complex format, but we only have index numbers – so how can we relate that back to actual file data?

External Object Records and the Blob Folder

Earlier, when describing the LevelDB keys that would allow us to identify records which contained objects stored by IndexedDB, we were looking for a key prefix containing the database ID, object store ID and then an index ID of 1 (then followed by the IdbKey structure containing the object's IndexedDB primary user key). We also noted that index IDs 2 and 3 also have special meaning, and when it comes to looking at files embedded in IndexedDB, it's index ID 3 – the "external object table" that we need to look into.

Every record in IndexedDB that contains embedded file data will have an associated external object table record. The embedded object table contains one or more IndexedDBExternalObject structures. Each of these structures provides metadata for a file embedded in IndexedDB. There is no count given in the data for the number of IndexedDBExternalObject structures, the intention is to just keep reading until you run out of data to read. Each of the structures can be enumerated, starting at 0; we spoke about FileIndex objects in the last section – those file indexes relate to the index of each IndexedDBExternalObject in the external object table for that record.

The data format has the following structure (documented in: https://github.com/chromium/chromium/blob/master/content/browser/indexed_db/indexed_db_backing_store.cc):

- Read a byte – should have value 0-2:
- If 0 – data is a blob
  - Read a varint – this is the "blob number"
  - Read a StringWithLength – this is the mime-type for the data
  - Read a varint – this is the length of the data
- If 1 – data is a file
  - Read a varint – this is the "blob number"
  - Read a StringWithLength – this is the mime-type for the data
  - Read a varint – this is the length of the data
  - Read a StringWithLength – this is the name of the file
  - Read a varint – this is the last modified time for the file (microseconds elapsed since 1601-01-01)
- If 2 – data is a native file handle (I haven't been able to generate these, more testing needed)
  - Read a blob (varint length followed by that many bytes of data) – this will be a "token"

So now we have the metadata for the files, but we still don't have the actual data itself. For that we need to make use of the "blob number" field from the `IndexedDBExternalObjects`.

If embedded files are present then alongside the "`indexeddb.leveldb`" folder for each domain there will be a matching "`indexeddb.blob`" folder, e.g. for "`https_docs.google.com_0.indexeddb.leveldb`" there will be a corresponding "`https_docs.google.com_0.indexeddb.blob`" folder.

To locate the data for a particular blob number, you will need to navigate to a path within the folder. The path references two pieces of data: the database id and the blob number. The path is made up of 3 parts:

- The database id
- The blob number in hex, omitting the final two digits. For this part of the path, the blob number is always assumed to be at least 16 bits in size and is padded with zeros if needed. E.g. blob number "270" is "010E" as a 16-bit hex number with padding, making this part of the path "01".
- The blob number in hex (no padding this time) – this is the file itself.

So, for blob number 270 in database ID 1 the path would be:

```
1/01/10E
```

This final step takes you from a FileIndex in the serialised object to the actual file data on disk.

The more you know

In this blog I've laid out how Chrome-esque apps encode their IndexedDB data into the LevelDB backing store (and a little bit on the filesystem). Elements of the individual formats can be found in other areas of Chrome et al: LevelDB especially, but the V8 serialisation format is also encountered in some other areas as well.

To help other researchers in this field we're excited to announce that we are open-sourcing the Python code that we generated during the research for this blog. In the set of scripts, you'll find pure Python implementations of: Snappy decompression, LevelDB, IndexedDB, V8 Deserialisation and Blink Deserialisation. The APIs still need some tidying up or wrapping to improve the coding experience, but they will do the job to get started digging into these files. Improvements will be on the way in the coming months and in the meantime if you find any bugs or missing features please feel free to submit a bug report or pull request!

The repository can be found here:

https://github.com/cclgroupltd/ccl_chrome_indexeddb