

Chapter 9

Protocol Buffers



Chris Currier

Abstract Protocol Buffers (Protobufs) are discussed in this chapter, from creating one to analyzing the data. This particular serialization format, originally developed by Google, is used in various apps. We discuss creating a protocol buffer and adding data through Python step by step. This provides a better understanding of how and why protocol buffers are formed and used. We also clarify how to recognize and decode them during a forensic examination.

9.1 Introduction

I remember being on the edge of my seat as Johan Persson, a developer at MSAB, first introduced me to Protocol Buffers. Why? *Protobufs*, as they are commonly referred to, contain data that we as examiners may find helpful in an investigation. I had no idea how to find them and view the payload they carried. However, that was to change quickly.

9.1.1 What is a Protocol Buffer?

A Protocol Buffer provides a format for taking compiled data (many different languages/platforms supported) and serializing it by turning it into bytes represented in decimal values. This makes the data smaller and faster to send over the wire. We call this *serialization* in computer science.

A protocol buffer is a data format structured in a very efficient binary format. The structure is defined in a *.proto* file, which is in a readable text format. The concept is similar to XML, where the schema description can be done inline or in a separate

Chris Currier

MSAB, Hornsbruksgatan 28 SE-117 34 Stockholm Sweden e-mail: chris.currier@msab.com

file. The *.proto* file is then used to generate code for reading from and writing data to the protocol buffer. Due to its nature, protocol buffer data is very suitable for transmission over networks. When transmitted over networks, it is often compressed with GZIP to minimize the size of the data. The protocol buffer concept was created and is used extensively by Google. Other users of protocol buffers include Apple and app developers.

So, where do we begin? Of course, the best place to learn about Google's Protocol Buffers is Google (see Fig. 9.1). Google defines how to structure and use Protocol Buffers [24]. The structured or rigid format used by Protocol Buffers is often referred to as a *schema*.

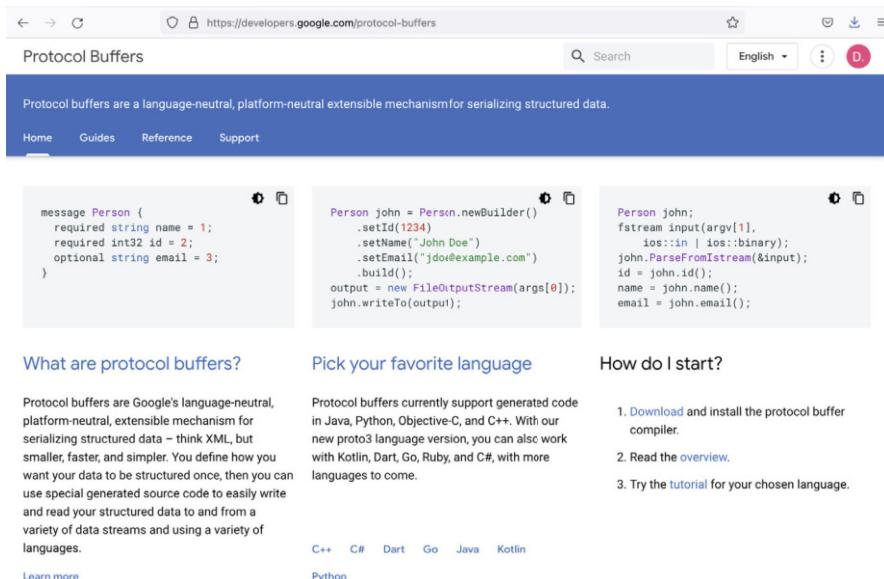


Fig. 9.1: Google's Protocol Buffers

Google itself says about its format in the developer documentation that it is a platform-independent, language-independent and easily extensible serialization format. Of course, other formats allow serialization of data too. Java-Serial is an example of this. Unlike this, Protobuf is a language-independent transmission format. Also, XML would be an option. However, Protobuf is smaller and faster than most of the other formats [24]. A significant advantage of Protobuf is that we only need to define the structure for the data to be transferred once and can then exchange it over a wide variety of data channels. The programming language is secondary since we are language-neutral. The data stream itself (network or file) is also irrelevant. The definition of the protobuf message always remains the same [24].

9.1.2 Why are Protocol Buffers Used?

Now think of a network with data being transmitted through it. How do we get data through the network faster? The smaller the data, the faster it will be. We also do not need it to be human-readable during transmission. This is where Protocol Buffers shine with faster transmission. Figures 9.2 and 9.3 demonstrate the time and size of Protobufs, based on tests performed to consider encoding and decoding benchmarks and common browsers [58, 43].

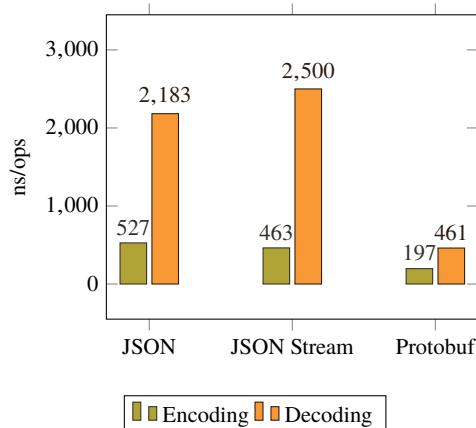


Fig. 9.2: Encoding and Decoding Performance of Protobufs [58]

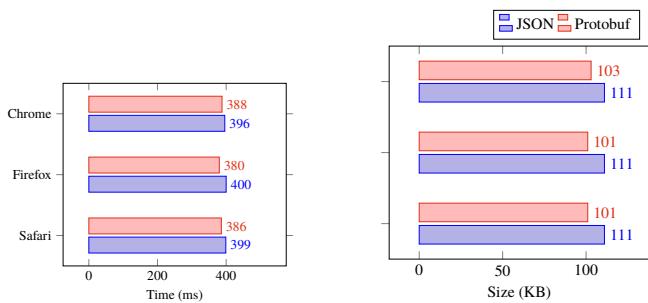


Fig. 9.3: Compression Environment of Protobufs and JSON [43]

The authors of a benchmark study in [43] conclude that ProtoBuf performs significantly better than JSON. The messages are significantly smaller and are transmitted

much faster at the same time. However, there is always someone faster, and that brings us to the term FlatBuffers:

“Protocol Buffers is indeed relatively similar to FlatBuffers, with the primary difference being that FlatBuffers does not need a parsing/ unpacking step to a secondary representation before you can access data, often coupled with per-object memory allocation. The code is an order of magnitude bigger, too. Protocol Buffers has neither optional text import/export nor schema language features.” [15]

As you can see, it is not just about speed but also the size of the data. The protocol buffer is not only the code but also the key. The data sent is binary and can be converted and looked at. Different languages (code) may be supported and used to enter and view this data.

9.2 Using Protocol Buffers

This section will clarify how ProtoBuf works and what data is needed. For this, the first step is to generate a description of the message types used and the access service:

Messages

To create a ProtoBuf message, we must first create an appropriate template. This template is usually saved in a file with the extension `.proto`. The file is set up and used alongside another programming language such as Python. The data (or user data) can be added using the same scheme (Field assignments: Type, Name, Tag) and then sent internally or externally over the wire. Google set up Protocol Buffers for their internal communication. Data is transmitted as binary. For this reason, we can encounter it in almost every Google app.

Services

Protocol Buffers are not just about messages but also services. For this reason, we need a service description. It describes the interface of the methods offered via the service. If we want to create an RPC service using a proto buffer, then the service must be given a name under which it can later be called once. In this case, the developer documentation recommends formulating both the service name and the access methods in CamelCase (with an initial capital). Here we have a brief example of defining a chat service that provides precisely one access method:

```
service ChatService {  
    rpc GetChats(ChatRequest) returns (CharResponse);  
}
```

One such service is Google's *gRPC*. These RPC (Remote Procedure Calls) methods accept a request "message" and return a response "message". Protobuf is often used with HTTP and RPCs for local and remote client-server communication. Protobuf is used for the description of the required interfaces and message types. The protocol composition is also summarized under the name *gRPC* [30]. In this case, the call to the remote method - encapsulated by a service - is provided platform-independently via a service description. The steps for generating a protobuf message and subsequent transmission are briefly summarised again in Fig. 9.4. The message data (message refers to type or object and not to chat or text) is then transmitted over the wire (internal or external). The supported platforms can all have code generated to deserialize the data and make sense of it, regardless of what coding platform created the data.

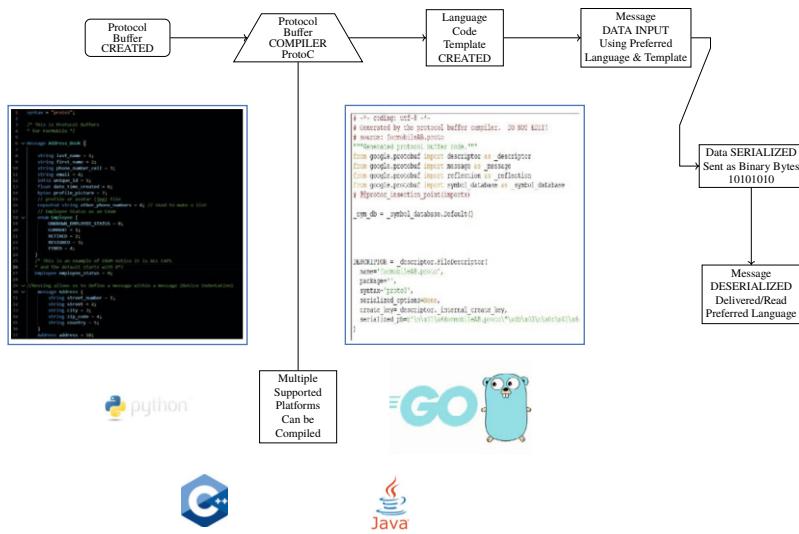


Fig. 9.4: How Protobufs Work

Looking at the data, we may not make sense of it without having the original code to know what the data represents. Unfortunately, we will not find the schema to help us make sense of the fields. So, let us look at the code to start with. That will give us a better understanding of what we see during the examination. Therefore, we will first discuss the design and implementation of a proto buffer using two small examples before turning to the forensic analysis of these special artefacts.

The Proto File

We start by creating a *.proto* file. We like to think of this file as the key or legend to the data we will see on a mobile device. Unfortunately, we will not find this key

or legend on the device itself as examiners. The `.proto` file is not included with the binary, with a few exceptions. As we will see, this often leaves some room for interpretation when we deserialize a Protocol Buffer manually. We will walk you through an example of the process. This example (see Fig. 9.5) will be for an Address Book, so we will name it `formobileAB.proto`. The file can be created with any editor.

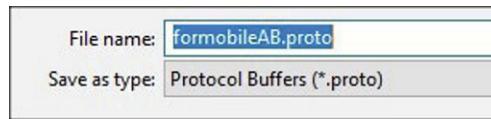


Fig. 9.5: FORMOBILE Protobuf Example

Define the Syntax

Google made Protocol Buffers public in 2008. In 2016 Google published Protocol Buffers 3. Since there is a different version, we have to identify which version of Protocol Buffers will be used. So, the first line of code needs to state this. Version 3 is used in these examples, as shown below.

```
syntax = "proto3";
```

Message Type

Now the message needs to be defined or named. This "message" is a code term that refers to the data and is not confused with terms chat or SMS text messages. The name should reflect the type of message based on content. The term *address book* should define our Protocol Buffer just fine. When using two words they use CamelCase and form one word i.e. `addressBook` or `AddressBook`, as shown below.

```
syntax = "proto3";  
  
message AddressBook {  
}
```

Fields

To create the first property, we need to know the type of data [int32], followed by property name [thread] and then identify it as the sequential property [1]. Fields

identify these data characteristics through Field Type, Field Name, and a Field Tag (or also called a *Field Number*). This is where we define the class characteristics that will be used. We consider what information would we want to know about, for example, a:

- Person
- Contact
- Web Browser Search
- Location and a Chat

! Attention

Remember, the idea behind Protocol Buffers is to take code from another language and package it into a smaller container. This starts defining the data by naming it and then following with fields.

The actual data such as: *John Smith 40 1.85 90.71 Brown Blue* will not exist in this proto file, but in another file. This should remind you of how meta data type looks like in a chat message. Fig. 9.6 is an example found in an address book showing the fields and associated data and profile picture.

Scalar Values

A message is normally composed of a number of different scalar values. Each value is assigned to a particular type. Looking at an SQLite database table definition, we will find terms such as Integer, Boolean, Float, and String. These define data types. Protocol Buffers use these as well (see table 9.1). Since we usually want to exchange messages between different applications, the data they contain must be preserved. As we can see in the table below, Protocol Buffers are easily used with other programming languages: C++, Java, Python, and Go. Accordingly, we can easily map the data type of a programming language to a ProtoBuf type and vice versa. For more information about types and unsigned bit integers please refer to [26].

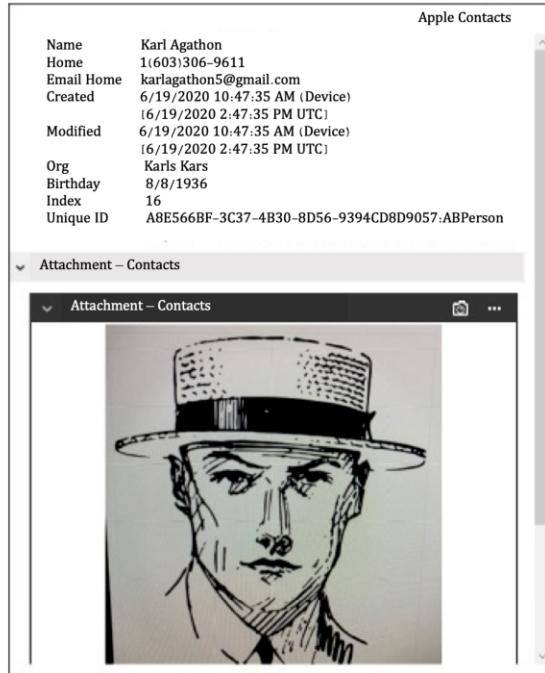


Fig. 9.6: Address Book Profile Example

.proto Type	Notes	C++	Java Type	Python Type	Go Type
double		double	double	float	*float64
float		float	float	float	*float32
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int	int	*int32
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64	long	int/long	*int64
uint32	Uses variable-length encoding.	uint32	int	int/long	*uint32
uint64	Uses variable-length encoding.	uint64	long	int/long	*uint64
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int	int	*int32
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	int64	long	int/long	*int64
sfixed32	Always four bytes.	int32	int	int	*int32
sfixed64	Always eight bytes.	int64	long	int/long	*int64
bool		bool	boolean	bool	*bool
string	A string must always contain UTF-8 encoded or 7-bit ASCII text.	string	String	unicode (Py2) or str (Py3)	*string
bytes	May contain any arbitrary sequence of bytes.	string	ByteString	bytes	[]byte

Table 9.1: Mapping Table for possible Scalar Types (Detail) [26]

! Attention

In this chapter, we will discuss several examples of Proto Buffers: (1) an address book, (2) a chat message, and (3) an Apple Maps example. You will find most of the files mentioned here: www.github.com/Xamnr/ProtocolBuffers. If you like, you can analyze the examples discussed here yourself. Just give it a try.

9.2.1 The Schema Definition

The Protocol Buffer defines the Object (type of data and position). Not the actual data or user data. The actual data will be coded in Python or another language format. However, the type of data that will go into these fields needs to be defined. Protocol Buffers use fields. There are three types of fields:

- Field Type
- Field Name
- Field Tag (or Number)

Field Type

The field type uses the scalar values to define the type of data like *integer*, *string*, or *bool*. Thinking back to our Apple contact, shown below (Fig. 9.7), we have the following fields and data to consider:

- Name or maybe Last Name and First Name
- Phone Number (Home, Work, Cell)
- Email (Home, Work)
- A Unique Identifier
- A Profile Picture

If not set, specified, or unknown, every field will have a default value. Protocol Buffers do not recognize required fields. Instead, the runtime environment of the programming language is responsible for that, i.e. Java, Python, Go. This means that a field whose assignment corresponds to the default value is not serialized. It is just left out. Since the field is missing in the data stream, the receiver side automatically uses the default value in this case. This property, which may seem confusing at first glance, ensures that no unnecessary values are transmitted. The serialized data on the wire remains small.

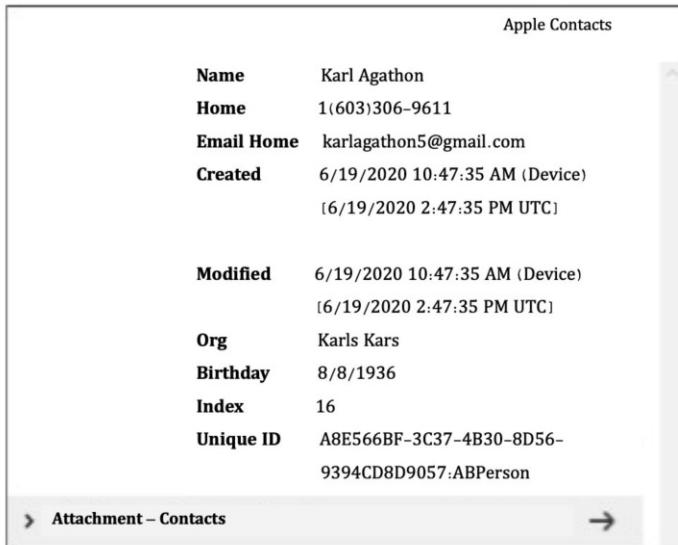


Fig. 9.7: Field Type and Scalar Values (example)

VALUE	DEFAULT
Bool	False
Number	0
String (UTF-8 or 7-bit ASCII)	Empty String
Byte (or byte array)	Empty Bytes (or empty byte array)
Enum	First Defined Value 0
Repeated	Empty List

Table 9.2: Field Type Default Values

Field Names

The field name represents one particular element within the message, therefore identifying the data for us. To identify a contact, we may use identifiers such as *last_name*, *first_name*, *phone_number_cell*, *email*, *unique_id*, *date_time_created*, *profile_picture*. When multiple words are used, each word is separated by an underscore '_'. Again, we do not add the data such as *Karl Agathon*. This data will be input elsewhere. Now that we have our field names, we need to identify the field type values for each field. We will focus on using Python. See Table 9.3 below.

The field tag is the last element. It works as a place holder. Tags are simply a number ranging from 1 to 536,870,911. However, there are some rules that come with these tags:

- The number may only be used once so that it is unique (more on this later).

VALUE	DEFAULT
last_name	String (UTF-8 or 7-bit ASCII)
first_name	String (UTF-8 or 7-bit ASCII)
phone_number_cell	Int (int32)
email	String (UTF-8 or 7-bit ASCII)
unique_id	Int (int32)
date_time_created	float (could be a double in another language)
profile_picture	Bytes

Table 9.3: Field Name Python Example

- Numbers 19000 through 19999 cannot be used. Reserved by Google.

There are also some strategies to speed up the data with these tags:

- Numbers 1 to 15 use only 1 byte, so these are used for fields used most often.
- Numbers 16 – 2047 use 2 bytes

Now we put this together in the code with the Field Type, Field Name, and Field Tag.



Fig. 9.8: Code Structure

A correct schema definition for our address book example could thus look like the following:

```
message AddressBook {
    string last_name = 1;
    string first_name = 2;
    int32 phone_number_cell = 3;
    string email = 4;
    int32 unique_id = 5;
    float date_time_created = 6;
    bytes profile_picture = 7;
```

As you can see above the address book has 7 assigned fields. Each field is defined by a Type, Name, and unique Tag. Certain rules still apply to the message fields [27]:

- *singular*: Such fields have the cardinality 1. Thus, a message can only have none or exactly one of this field values. This is the default rule.

- *repeated*: Array or list of values. It can be repeated any number of times - even zero times.

Since the keyword *singular* is the default case, it can be omitted when defining a field. Once we have chosen the field tag number, that number is unique and cannot be reused. However, we could change the *.proto* file by commenting out a field. Field names or field tags can also be reserved for future use. Using a reserved field may cause compiler issues if the data type is not identified correctly.

For example, to define the field other phone numbers as a list, we can use the following assignment:

```
repeated string other_phone_numbers = 8;
```

Enums

An *Enumeration (Enum)* is used when the values for a field are known or fixed. An Enum must start with tag 0 (default value). An example could be a status: Unknown Status (default), Read, Unread, Sent. The Enum values are all capitalized (upper case). See the example below. Here we added the employee's employment status. The default value is the first one tagged with zero:

```
1 syntax = "proto3";
2
3 /* This is Protocol Buffers
4 * for FORMOBILE */
5
6 message AddressBook {
7     string last_name = 1;
8     string first_name = 2;
9     int32 phone_number_cell = 3;
10    string email = 4;
11    int32 unique_id = 5;
12    float date_time_created = 6;
13    bytes profile_picture = 7;
14    // profile or avatar (jpg) file
15
16    repeated string other_phone_numbers = 8;
17
18    //Employee Status as an Enum
19    enum EmployeeStatus{
20        UNKNOWN_EMPLOYEE_STATUS = 0;
21        CURRENT = 1;
22        RETIRED = 2;
23        RESIGNED = 3;
24        APPLICANT = 4;
25        FIRED = 5;
26    }
27
```

```
28  /* This is an example of an ENUM notice
29  * it is ALL CAPS and the default starts with zero */
30  EmployeeStatus employee_status = 9;
31 }
```

Nesting

Messages can be added inline into another message. Nesting allows us to have message(s) types within a message type. This functionality is well known in programming languages and is called aggregation. That means some other message type is part of a second message type.

! Attention

Here a message refers to code and not a chat message.

In the example shown below, the address entry has been added to include street, city, zip code, and country. Notice the indentation. In this example, the original message refers to the `AddressBook`, and the nested message refers to the message `Address` that starts on line30. The `enum` used tag 9 and the nested `message` is now assigned tag 10. This is now defined as `AddressBook.Address`.

```
1 syntax = "proto3";
2
3 import "myproject/timestamp.proto";
4
5 message AddressBook {
6     string last_name = 1;
7     string first_name = 2;
8     int32 phone_number_cell = 3;
9     string email = 4;
10    int32 unique_id = 5;
11
12    //....
13
14    //Nesting allows us to define a message within a
15    //message (notice the indentation)
16    message Address{
17        string street_number = 1;
18        string street = 2;
19        string city = 3;
20        string zip_code = 4;
21        string country = 5;
22    }
23    Address employee_address = 10;
24 }
```

Importing & Packages

Importing allows us to use other .proto file(s) or package(s) with the code you need from a different proto file. Below is a timestamp.proto file that has the set up for an epoch time stamp, which will be shown on the following pages.

When importing, we use *import* followed by the full path where the file is located ending with a semicolon, as seen below. Code can be compiled and put into a package. Protocol Buffers are no different, which is helpful for other coding languages. This also helps to avoid naming conflicts. A package can be created and then imported into a protocol buffer. Following is the timestamp.proto file. The package name is google.protobuf.timestamp and save the file to the same directory as the formobileAB.proto file.

```

syntax = "proto3";

package google.protobuf.timestamp;

option csharp_namespace = "Google.Protobuf.WellKnownTypes";
option cc_enable_arenas = true;
option go_package = "github.com/golang/protobuf/ptypes/timestamp";
option java_package = "com.google.protobuf";
option java_outer_classname = "TimestampProto";
option java_multiple_files = true;
option obj_class_prefix = "GPB";

```

```

message Timestamp {
    // Represents seconds of UTC time since Unix epoch
    // 1970-01-01T00:00:00Z. Must be from 0001-01-01T00:00:00Z to
    // 9999-12-31T23:59:59Z inclusive.
    int64 seconds = 1;
    // Non-negative fractions of a second at nanosecond resolution.
    // Negative second values with fractions must still have
    // non-negative nanos values that count forward in time.
    // Must be from 0 to 999,999,999 inclusive.
    int32 nanos = 2;
}

```

To include the message definition of a Timestamp to our address book, we have to open formobileAB.proto file and add the imported proto file as well as the package name. Now we have to change the ‘date_create’d field so that the timestamp epoch time is recognized from the package:

```

1 syntax = "proto3";
2
3 /* This is Protocol Buffers
4 * for FORMOBILE */
5 import "google/protobuf/timestamp.proto";
6

```

```

7 package google.protobuf.timestamp;
8
9 message AddressBook {
10   string last_name = 1;
11   string first_name = 2;
12   int32 phone_number_cell = 3;
13   string email = 4;
14   int32 unique_id = 5;
15   google.protobuf.timestamp.Timestamp date_created = 6;
16   bytes profile_picture = 7;
17   // profile or avatar (jpg) file

```

Now we have some idea of how a protocol buffer .proto file is created. The .proto file itself does not contain user data but just the schema. We will find such schema definitions in our analysis. However, they may appear like the timestamp.proto file shown in Fig. 9.9. Our analysis tool or a hex viewer may not be the best way to view this file. Here, an ordinary text editor is certainly the better choice.

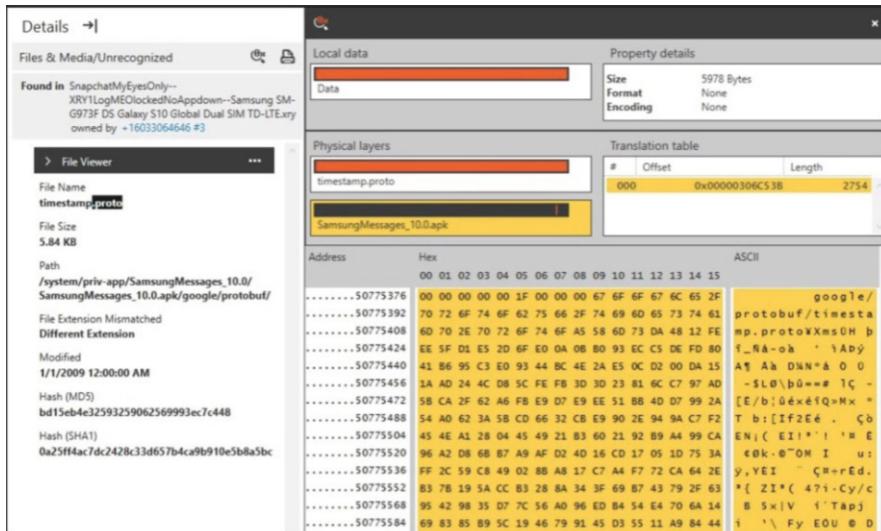


Fig. 9.9: timestamp.proto demonstrated in a Hex Viewer

The above information covers some of the code options for creating the .proto file. More information can be found at [27]. We will now look at the .proto file from a forensic analysis perspective. But first, we have to transfer our newly created message type into a concrete programming language.

9.2.2 Compiling Your Protocol Buffer

Once the custom data structures are defined as desired in the .proto file, generate the classes needed to read and write the protobuf messages. For this purpose, apply the protocol buffers compiler (protoc) to the configuration file. The `protoc.exe` is what we will be using to look at the data that we find. So first, let us see how it is used to serialize or encode the data from a protocol buffer file. The link to obtain ProtoC is [www.github.com/protocolbuffers/protobuf/releases](https://github.com/protocolbuffers/protobuf/releases). ProtoC will generate code from the Proto File to the supported language. A template for coders to follow and use the defined terms.

First, we have to specify the directory to search for imports. It may be specified multiple times; directories will be searched in order. If not given, the current working directory is used. If not found in any of these directories, the `--descriptor_set_in` descriptors will be checked for required proto file. Next, we have to define the output language:

<code>--cpp_out=OUT_DIR</code>	Generate C++ header and source.
<code>--csharp_out=OUT_DIR</code>	Generate C# source file.
<code>--java_out=OUT_DIR</code>	Generate Java source file.
<code>--js_out=OUT_DIR</code>	Generate JavaScript source.
<code>--objc_out=OUT_DIR</code>	Generate Objective-C source.
<code>--php_out=OUT_DIR</code>	Generate PHP source file.
<code>--python_out=OUT_DIR</code>	Generate Python source file.
<code>--ruby_out=OUT_DIR</code>	Generate Ruby source file.

In this case, we will be using Python `--python_out=OUT_DIR`. Other languages like GO are supported and can be found referenced online. Now to take the `formobileAB.proto` file and compile the code for Python (or another language). We will place the ProtoC executable here and create a python folder.



Fig. 9.10: Python Folder Example

Then open up a command prompt in this location and follow steps 1, 2, 3 or 1, 2, 4.

- 1 Determine the directory name that your proto files are in.
- 2 Add Output language (Java, Python...).
- 3 Add Absolute path of your proto file with extension.
- 4 or all proto files in that location folder.



```
$>protoc --proto_path=. --python_out=./formobileAB ./formobileAB.proto
```

Fig. 9.11: File Path Example

Analysing the Python Protobuf-Code

In our example, we have chosen Python as the target language. We will briefly discuss the file `formobileAB_pb2.py` created in the process below. In the first section, we see imports from `google.protobuf`. One of the imports mentions reflection. This can be observed throughout the following Python example. This means the coder will have to identify the objects in their code. Descriptors are shown as well. A `serialized_pb` binary buffer could be found.

```
# -*- coding: utf-8 -*-
# Generated by the protocol buffer compiler. DO NOT EDIT!
# source: formobileAB.proto
"""Generated protocol buffer code."""
from google.protobuf import descriptor as _descriptor
from google.protobuf import message as _message
from google.protobuf import reflection as _reflection
from google.protobuf import symbol_database as _symbol_database
# @@protoc_insertion_point(imports)

_sym_db = _symbol_database.Default()

DESCRIPTOR = _descriptor.FileDescriptor(
    name='formobileAB.proto',
    package='',
    syntax='proto3',
    serialized_options=None,
    create_key=_descriptor._internal_create_key,
    serialized_pb=b'\n\x11\x66ormobileAB.proto"\xf7\x03\n\x0b\x41\x64\x64ressBook\x12\x11\n\x18\x01 \x01(\t\x12\x12\n\n\x18\x02(\t\x12\x19\n\x11phone_number_cell\x18\x03 \x01(\x05\x12\r\n\x05\x65mail\x18\x04 \x01(\t\x12\x11\n\x18\x05 \x01(\x05\x12\x14\n\x0c\x64\x61te_created\x18\x06 \x01(\x02\x12\x17\n\x0fprofile_picture\x18\x07 \x01(\x0c\x12\x1b\n\x13other_phone_numbers\x18\x08 \x03(\t\x12\x34\n\x0f\x65mployee_status\x18\t\x01(\x0e\x32\x1b.AddressBook.EmployeeStatus\x12.\n\x10\x65mployee_address\x18\n\x01(\x0b\x32\x14.AddressBook.Address\x1a\x61\n\x07\x41\x64\x64ress\x12\x15\n\x18\x01 \x01(\t\x12\x0e\n\x06street\x18\x02 \x01(\t\x12\x0c\n\x04\x63ity\x18\x03 \x01(\t\x12\x10\n
```

```
\x08zip_code\x18\x04 \x01(\t\x12\x0f\n
\x07\x63ountry\x18\x05 \x01(\t\"o\n
\x0e\x45mployeeStatus\x12\x1b\n
\x17UNKNOWN_EMPLOYEE_STATUS\x10\x00\x12\x0b\n
\x07\x43URRENT\x10\x01\x12\x0b\n\x07RETIRED\x10\x02\x12\x0c\n
\x08RESIGNED\x10\x03\x12\r\n\tAPPLICANT\x10\x04\x12\t\n
\x05\x46IRED\x10\x05\x62\x06proto3'
)
```

Scrolling down the page we find the `AddressBook` message descriptors. You should be able to see the Field Names and the Field Tags.

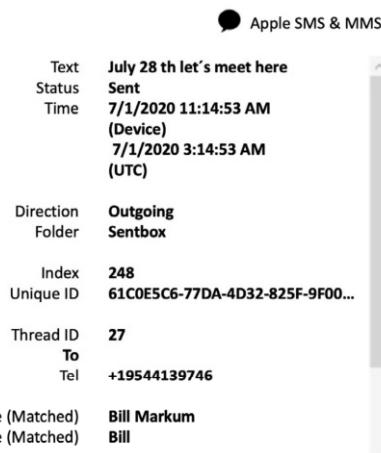
```
full_name='AddressBook.Address',
filename=None,
file=DESCRIPTOR,
containing_type=None,
create_key=_descriptor._internal_create_key,
fields=[
    _descriptor.FieldDescriptor(
        name='street_number',
        full_name='AddressBook.Address.street_number',
        index=0, number=1, type=9, cpp_type=9, label=1,
        has_default_value=False, default_value=b"".decode('utf-8'),
        message_type=None, enum_type=None, containing_type=None,
        is_extension=False, extension_scope=None,
        serialized_options=
        None, file=DESCRIPTOR, c
        reate_key=_descriptor._internal_create_key),
    _descriptor.FieldDescriptor(
        ←
        name='street', full_name='AddressBook.Address.street', index=1,
        number=2, type=9, cpp_type=9, label=1,
        has_default_value=False, default_value=b"".decode('utf-8'),
        message_type=None, enum_type=None, containing_type=None,
        is_extension=False, extension_scope=None,
        serialized_options=
        None,
        ←
        file=DESCRIPTOR,   create_key=_descriptor._internal_create_key),
```

A 2nd Example - The `FormobileChat` message

Having created the first example so easily, let us follow it up with a second example right away. This time it will be about defining a chat message with data fields and then generating a corresponding protobuf message. A second example was generated. The `formobilechat.proto` file has been created. After reading through this material you should have a good idea of what you are looking at. There is a message named `FormobileChat`. Followed by Field Types, Names, and Tags. There are also two enums

used for message direction and status. Does this data remind you of something? Chat message data, maybe?

```
syntax = "proto3";
// Formobile Protocol Buffers
message FormobileChat {
    int32 chat_thread_id = 1;
    string chat_contact = 2;
    string chat_text = 3;
    bytes chat_attachment = 4;
    float chat_latitude = 5;
    float chat_longitude = 6;
    int64 chat_timestamp = 7;
enum Chat_Direction {
    UNKNOWN_DIRECTION = 0;
    OUTGOING = 1;
    INCOMING =2;
}
Chat_Direction chat_direction = 8;
enum Chat_Status {
    UNKNOWN_STATUS = 0;
    UNREAD = 1;
    READ = 2;
}
Chat_Status chat_status = 9;
```



As with our address book example, next, we need to have the schema file compiled using the compiler protoc. The result, in our case, is again a Python source file. We could use ProtoC -proto_path and -python_out commands to generate the code for Python.

! Attention

Note there are two dashes “–” before both proto and python.

```
$> protoc --proto_path=. --python_out=. ./formobilechat.proto
```

ProtoC took the `proto` file, output it to Python to create the `formobilechat_pb2.py` file. This file has almost 200 lines of code from a `proto` file with less than 30 lines of code.

`Formmobilechat_pb2.py`

Even though it says `pb2`, this was made from a `proto3` file. Notice the size compared to the `.proto` file itself.

```
# -*- coding: utf-8 -*-
# Generated by the protocol buffer compiler. DO NOT EDIT!
# source: formobileAB.proto
"""Generated protocol buffer code."""
from google.protobuf import descriptor as _descriptor
from google.protobuf import message as _message
from google.protobuf import reflection as _reflection
from google.protobuf import symbol_database as _symbol_database
# @@protoc_insertion_point(imports)

_sym_db = _symbol_database.Default()

DESCRIPTOR = _descriptor.FileDescriptor(
    name='formobileAB.proto',
    package='',
    syntax='proto3',
    serialized_options=None,
    create_key=_descriptor._internal_create_key,
    serialized_pb=b'\n\x11\x66ormobileAB.proto"\xf7\x03\n\x0b\x41
\x64\x64ressBook\x12\x11\n
\tlast_name\x18\x01 \x01(\t\x12\x12\n\n
\tname\x18\x02(\t\x12\x19\n
\x11phone_number_cell\x18\x03 \x01(\x05\x12\r\n
\x05\x65mail\x18\x04 \x01(\t\x12\x11\n
```

9.2.3 Creation of a Protobufs with Python

Now it is time to generate our first chat message using the Python files generated in the previous step. Therefore, a python script must be created, so this one will be named `formobilechat.py`.

First, the `formobilechat_pb2` has to be imported into the script and followed by any other imports or packages. Without the import, we would not be able to access the message types predefined. A variable is created, identifying the Fieldnames and entering data for those fields. Since *Reflection* is used, the developer must identify the fields used in the Protocol Buffer. In programming, reflection means that a programme knows its structure (introspection) and can modify it.

Program Code <formobilechat.py>

```
import formobilechat_pb2 as formobilechat_pb2

FormobileChat = formobilechat_pb2.FormobileChat()

FormobileChat.chat_thread_id = 1
FormobileChat.chat_contact_id = "Karl Agathon"
FormobileChat.chat_text = "Patrick sorry you could not make it
    tonight to get your cut of the cash. We will use you for the
    next bank. Got something for you"
FormobileChat.chat_attachment = bytes([0xFF, 0xD8, 0xFF, 0x00,
    0x10, 0x4A, 0x46, 0x49, 0x46, 0x00, 0x01, 0x01, 0x01, 0xFF,
    0xD9])
FormobileChat.chat_latitude = 5.50559
FormobileChat.chat_longitude = -0.08956
FormobileChat.chat_timestamp = 1616182435
FormobileChat.chat_direction = 1
FormobileChat.chat_status = 2
```

In our example, a chat message is generated with a Contact named *Karl Agathon*. In addition to the actual message text, a JPEG was also added as an attachment. The message is supplemented with position information (latitude and longitude) and a timestamp. Now that the sample message is complete, we can create a real ProtoBuf message from it in the next step.

Writing the Object to a Binary File

The message is now serialized using Protobuf and saved to a binary file. For this we create a new file named *FormobileChat.bin*. Then we write the content of the messages created with Python before into the file.

Program Code

```
with open("FormobileChat.bin", "wb") as f:
    bytesAsString = FormobileChat.SerializeToString()
    f.write(bytesAsString)
```

The output file is then located in the same directory as the Python script used to create the binary.

Remember Size = Speed

Notice the size comparisons below. The first image shows the Python Script `formobilechat.py` and the `FormmobileChat.bin`. This contains the complete `chat_attachment jpg` picture binary data.

Name	Size	Type
FormmobileChat.bin	39 KB	BIN File
formobilechat.py	232 KB	Python File

Fig. 9.12: FormmobileChat.bin in File Explorer

Notice the size comparison of the original proto file, the compiled `pb2.py` file, and the binary file. Note the `FormmobileChat.bin` (has the full jpg picture `chat_attachment` binary data). The `FormmobileChatsmall.bin` has a portion of the `chat_attachment` binary data as seen on the previous page.

Name	Size	Type
__init__.py	0 KB	Python File
FormmobileChat.bin	39 KB	BIN File
formobilechat.proto	1 KB	PROTO File
formobilechat.py	1 KB	Python File
formobilechat_pb2.py	9 KB	Python File
FormmobileChatsmall.bin	1 KB	BIN File

Fig. 9.13: FormmobileChat.bin in File Explorer

The Raw Binary Data

Opening the file in Hex-Editor does not really do this file justice. Well we can see the chat text and the file signature of a JPG, but that is it. So how do we handle this protocol buffer data?

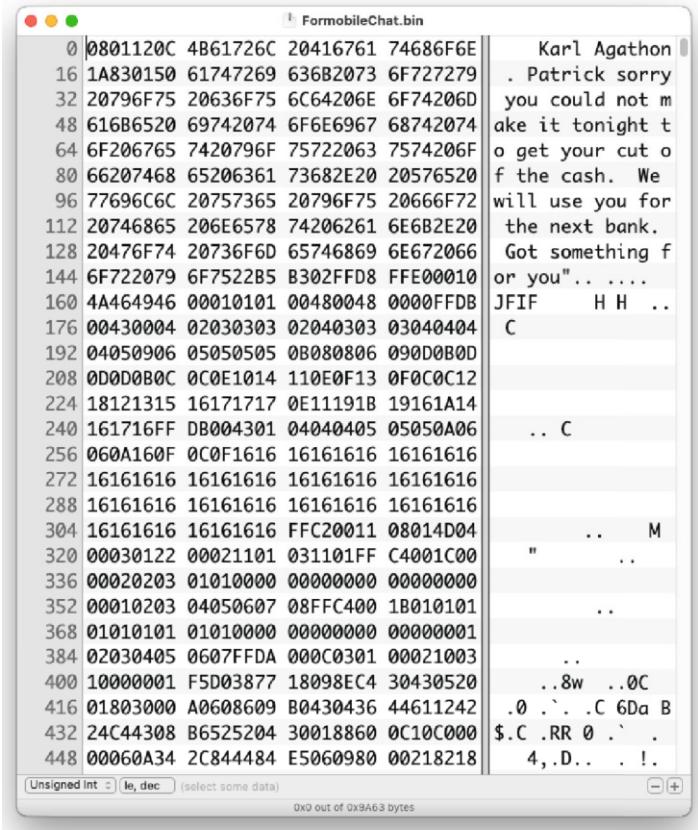


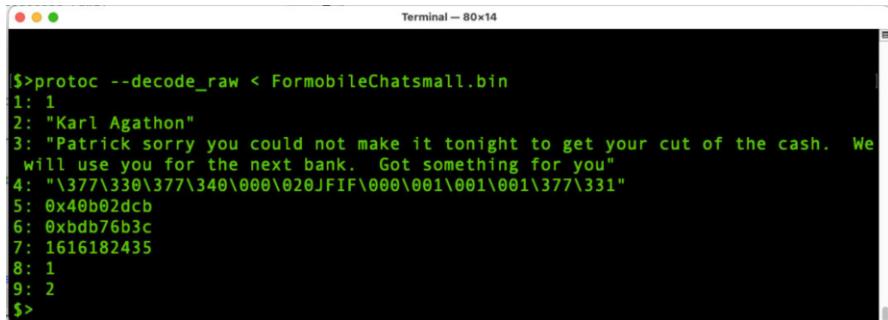
Fig. 9.14: FormobileChat.bin Hex

9.2.4 Reversing Proto Buffer Messages

In our example, we are in possession of the original *.proto* file as well as the generated binary. In practice, unfortunately, it is often the case that we do not have a schema file. But even without a interface description there is a way out.

The `protoc` compiler is not just for compiling data from a protocol buffer. But it can also be used in other ways. The `protoc` tool is very useful for showing the contents of protocol buffer data.

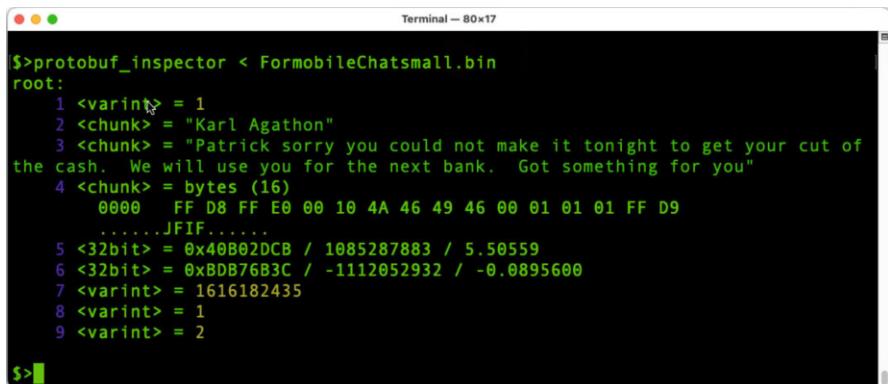
There is data here for us to find. We just need to know how to view it. That is where the `protoc -decode_raw` command comes in. We use the command line to decode the raw binary data from the `FormobileChatsmall.bin`. The command to use is `protoc --decode_raw < (File and Path)`. Our attempt to restore the data using the "decode raw" option was apparently only partially successful (see Fig. 9.15).



```
$>protoc --decode_raw < FormobileChatsmall.bin
1: 1
2: "Karl Agathon"
3: "Patrick sorry you could not make it tonight to get your cut of the cash. We
will use you for the next bank. Got something for you"
4: "\377\330\377\340\000\020JFIF\000\001\001\001\377\331"
5: 0x40b02dcb
6: 0xbdb76b3c
7: 1616182435
8: 1
9: 2
$>
```

Fig. 9.15: Decoded Protocol Buffer

Fortunately, there is a solution for this as well. Thus, there are a variety of programs that provide a mostly accurate interpretation of the numerical values. The program *protobuf-inspector*¹ is one of those tools. It helps to reverse Protocol Buffers with unknown definition, i.e., missing .proto files. The command to use is `main.py < (File and Path)`



```
$>protobuf_inspector < FormobileChatsmall.bin
root:
  1 <varint> = 1
  2 <chunk> = "Karl Agathon"
  3 <chunk> = "Patrick sorry you could not make it tonight to get your cut of
the cash. We will use you for the next bank. Got something for you"
  4 <chunk> = bytes (16)
    0000 FF D8 FF E0 00 10 4A 46 49 46 00 01 01 01 FF D9
      .....JFIF.....
  5 <32bit> = 0x40B02DCB / 1085287883 / 5.50559
  6 <32bit> = 0xBDB76B3C / -1112052932 / -0.0895600
  7 <varint> = 1616182435
  8 <varint> = 1
  9 <varint> = 2
$>
```

Fig. 9.16: Close up of Protobuf-Inspector decode results

With both decodes, we see the data entered, and some of it is easily understood, and other parts are not. Take note above that the *protobuf inspector* does change the octal values to hexadecimal and also translated the Longitude and Latitude in the correct Decimal 5.50559 and –0.0895600. We can compare the encoded binary message with the original Python Script results (see Fig. 9.17).

¹ www.github.com/mildsunrise/protobuf-inspector

```

formobilechat ×
chat_thread_id: 1
chat_contact: "Karl Agathon"
chat_text: "Patrick sorry you could not make it tonight to get your cut &
chat_attachment: "\377\330\377\340\000\020JFIF\000\001\001\377\331"
chat_latitude: 5.50559
chat_longitude: -0.08956
chat_timestamp: 1616182435
chat_direction: OUTGOING
chat_status: READ

```

Fig. 9.17: The Original Entered Formobilechat Data (Python)

Data Conversion

While we may not know what the 1 and 2 flags mean, we can certainly look for data that we can do something about to start with. Location data in the case of 5.50559 and -0.08956 is shown in decimal, which is one way forensic tools represent it. However, what do we do when we are seeing something (maybe from a map application) that could be longitude and latitude and is in Hexadecimal: 0x40b02dc_b 0xbdb76b3c If the Hex value starts with 8, 9, A, B, C, D, E, or F then it is a negative number. There are a few ways to do this, but the best we have been taught is HxD and a Python Script. We will show you how to use HxD's Data Inspector later in the chapter.

The Python Script requires 8 bytes, as seen below and do not name the script `struct` (as that is reserved). Also, be aware that you may input the data in the wrong spots mixing up the latitude and longitude. Test this with known data first to make sure it works in your part of the world.

Program Code

```

#convert Lat Long from hex to decimal
import struct

lat = struct.unpack('>d', b'\x40\x45\xF5\xE5\xF6\x6D\x59\x0F')[0]
long = struct.unpack('>d', b'\xc0\x51\xFA\xA3\xB1\xD3\x4B\x67')[0]

print("Latitude: ", (lat), "and the Longitude: ", (long))

```

Timestamp

In the above example, we see something that may be an epoch timestamp 1616182435. Unix time is based on the date 1970-01-01 00:00:00 (UTC), and Apple timestamps

(MAC Absolute time) use 2001-01-01 00:00:00 (UTC) as a start. Timestamps normally use seconds but may also use milliseconds, microseconds, nanoseconds etc.

The above example starts with 1 then it is probably a Unix time (when it comes to Android and Apple Devices). Apple Timestamps will most likely start with 3, 4, 5, or 6. We find epoch converter works well as an online converter or `Tempus.pyw` for an offline converter available at <http://github.com/eichbaumj/Python>.

Linux	CF or Mac Absolute	Apple HFS+
www.epochconverter.com	www.epochconverter.com/coredata	www.epochconverter.com/mac

Table 9.4: Epoch Timestamp Look Up Websites

Conversion of UNIX Epoch Time: **1616182435**

Date and Time:	Friday, 2021-03-19 19:33:55 UTC
UNIX Epoch Time:	1616182435
UNIX Epoch Time (Hex):	6054FCA3
CF Absolute Time:	637875235

Fig. 9.18: Epoch Timestamp Look Up

Pictures or other files represented by octal data

The easiest way is to look at the data in a Hex Viewer such as HxD or your forensic tool and copy out the file. In this case, the attachment is a JPG. The file signature for a JPG file is Hex FFD8FF, and the end of the file may have Hex FFD9. You can see the start of the file below. Highlight and copy the data and save it as a .jpg type file. Example files can be found here: <https://github.com/Xamnr/ProtocolBuffers>.

9.3 Practical Analysis of different Proto Buffers

Analyzing digital evidence when looking thoroughly at an application can be difficult enough. Within a normal investigation of a mobile phone, the investigator already has to evaluate many different file formats. Typical file formats, which are also found in this book, are Apple Property Lists (Plists), XML, SQLite databases. Of course, you can do keyword searches for .proto files, but as you saw earlier, that is probably

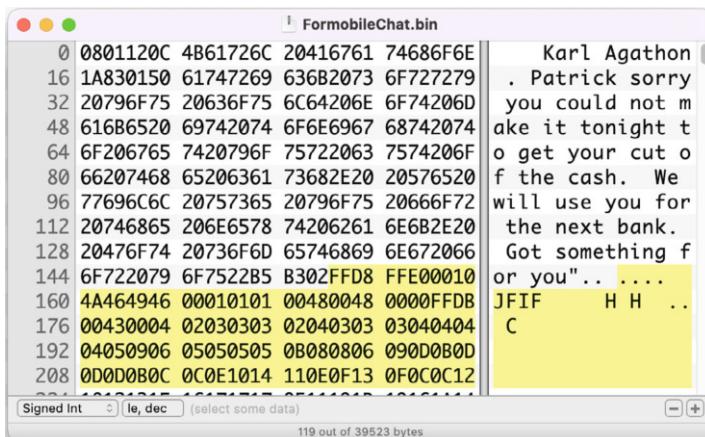


Fig. 9.19: ChatData - HexView, including file header of JPEG picture

not going to help you. To make even worser, proto buffers are often nested within other file formats rather than in their own files.

One of the issues is that these files already contain various types of data, such as Binary Large Object encoded Base 64. So, we need to be familiar with another XML or PList file within a BLOB, XML, Plist and/or Protocol Buffer Data within a BLOB. BLOBS are often also stored in a database table. In each case, we have to determine what content we are dealing with. Unfortunately, Protobuf does not have a real MagicNumber. We can only make a guess. Even more, Protocol Buffers can also be stored as GZip archive files. This adds another level of difficulty in finding these Protocol Buffers.

9.3.1 Mobile Device Artifact Examples

Some popular apps that use protocol buffers include Apple Maps, Google Maps, Badoo, Gmail, Google Allo, TamTam, Tango, WeChat, Wickr, Wire and many more. The Apple iCloud Backup system makes extensive use of protocol buffers. When dealing with application data, you are probably familiar with SQLite Databases, XML, and Apple Property List Files (.plist). You may not be aware that these files can contain data encoded Base64, such as a Binary Large Object (BLOB).

Example - Waze Navigation App

As a first example, we will use a typical app that uses Proto Buffers internally. *Waze* is a navigational guidance application for getting directions and showing the fastest

available routes. Shown on the following page is the application folder for Waze. Selected is a file named `cache_data`, shown below.

> File Viewer	
File Name	cached_data
File Size	476 Bytes
Path	/private/var/mobile/Containers/Data/Application/com.waze.i phone/Documents/
File Extension Mismatched	No Extension
Created	2/11/2021 11:08:35 AM (Device) [2/11/2021 4:08:35 PM UTC]
Modified	2/15/2021 9:55:26 PM (Device) [2/16/2021 2:55:26 AM UTC]
Status Changed	2/15/2021 9:55:26 PM (Device) [2/16/2021 2:55:26 AM UTC]
Hash (MD5)	1303b0fff1558c5a465da38ad843b8c
Hash (SHA1)	40311eb40eb6f237aed38134a7d0896034b87b00

Fig. 9.20: `cached_data`

Let us now take a closer look at the cache file. We use the forensic tool's hex viewer to see what type of data the file contains, shown on the right. Well this is nice, we can see some data immediately (see Fig.9.21). With a simple string search we can already extract a number of artefacts. A couple things you should know about the data (see Table 9.5).

Username: Millenium Falcon
Phone Number: +15166618197
Home Location: 375 Main Street, New London, NH 03257

Table 9.5: Some extracted data

In fact, this example is a protocol buffer. We save the file out as a binary file adding the `.bin` file extension. Now to examine the file with both *protoc* and *protobuf inspector*. In Fig. 9.22 Waze's `cache_data` decoded with ProtoC. Scrolling down through the results, we come across the address and again some other data that we may or may not determine. You may not figure out what the other data items are. Again, we do not have the original code or legend. We can certainly see if data is a timestamp or location.

Below in Fig.9.23 we take the same file into **Protobuf Inspector**. As we can see, some characters on the screenshots are from escape sequences to make sure that some chars (characters) are in bold, etc. This is an easy example, in my opinion, of a protocol buffer. Using *Protoc*, *Protobuf Inspector*, or other Protocol Buffer decoding tools can help break down and show the information. In this case, the data could



```
Terminal - 95x20
$>protoc --decode_raw < cached_data.bin
1 {
  1: "Millenium"
  2: "Falcon"
  3: "+15166618197"
  5: 0
  6: ""
  7: 0
  10: 150528866/
  11: 59191
  12: 1613072688
  14: 1
  15: 1
}
2{
  3: "https://profileimages-na.waze.com/SocialMediaServer/images/profile/fb44cf36-a989-47ca-a1
  4{
    1: "contacts"
}:
```

Fig. 9.21: cached_data

be seen for the most part with the hex viewer. Other instances will contain Base64 Encoded data.

BASE64 Encoding

Some of you may remember reading about BASE64 and its use with Email Attachments. This encoding scheme is to take this raw data like a picture and make sure that none of the data will cause an issue. Looking at an ASCII chart, you will notice the first 33 decimal places (0-32) are reserved for functions like BackSpace, Space, and Carriage Return.

! Attention

Please remember that Apple Property Lists, XML Files, and Databases can contain pictures or web links to pictures, and of course, Protocol Buffers. Next, we will analyze some examples of Protocol Buffers found in each.

When we send raw data, we do not want these functions to be performed. So Base64 Encoding removes these from the equation. Binary Large Objects (like a picture or even an embedded XML or Plist file) are encoded with Base64. So, what are Protocol Buffers managing? Raw data. When it comes to plists, they are usually in binary form, only rarely in text format. A forensic tool like MSAB's XAMN will show such content in a readable (XML-like) format, making binary data appear as base64. The website used in the below example to convert the raw data to Base64: www.motobit.com/util/base64-decoder-encoder.asp.



```
ASCII
0 ; Millenium
Falcon +15166
618197( 2 8 P E
I X-I `` p x
L https://pro
fileimages-na.wa
ze.com/SocialMed
iaServer/images/
profile/fb44cf36
-a989-47ca-a1f0-
b9296fb80da6" P
contacts"@d9f117
0b6404cde3300963
ba61bd64e78febfb8
fc5d7f40e28dd099
d2c5580a6a8 @ (
H " " R
Q NS
" " | Ø
÷: " þÃØÝÝÝÝÝÝ
" öâÙ 375 Main
St" Main Street
" New London2 NH
: B 375J(googleP
laces.ChiJ_4k6x5
b44YkRjbQDEY0wBk
0P " Ù Z Home
" (E uÃi. @ Hï
~ð P#Z 1|global
|504040814"
```

Fig. 9.22: cached_data

```
Terminal — 48x9
$>protobuf_inspector < cached_data.bin
root:
    1 <chunk> = message:
        1 <chunk> = "Millenium"
        2 <chunk> = "Falcon"
        3 <chunk> = "157166618197"
        5 <varint> = 0
        6 <chunk> = empty chunk
:
```

Fig. 9.23: cached_data in Protobuf Inspector

Example: Apple Web Cache file

In Fig. 9.24 you can see a Apple Web Cache file. The filename is *I2.xml*. The BLOB is highlighted. But what is it about in this case? Is it a protocol buffer or something else?

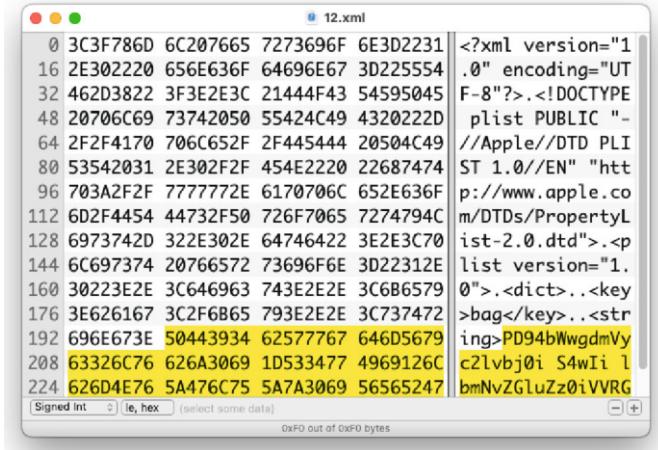


Fig. 9.24: 12.xml File Highlighting Encoded Data

We get the answer when we convert the raw data - probably BASE64 encoded - back into a normal UTF-8 string. The result is shown in Fig. 9.25. The BASE64 converted data, and we will notice that this is an XML file within an XML File. What can we learn from this? It does not always have to be a protobuf.

```
Source data from the Base64 string:
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>refresh_cycle</key>
    <integer>86400</integer>
    <key>release</key>
    <dict>
        <key>gonzo</key>
        <dict>
            <key>clients</key>
        </array>
```

Fig. 9.25: 12.xml File Base 64 Converted Data

Identifying Base64 Encoded Data

You know that Binary Large Objects (BLOBS) are usually, if not always, encoded with BASE64. Sometimes you do not know. As seen below, we look for the tell-tale equals sign “=” or two equal’s signs “==” at the end of the data.

! Attention

Note the BASE64 encoded data does not always end with an equals sign.

```
C BYQACDA0QI oADAAYAKBfbWIbIcCwsJBKK5NOicKJQiL84igk82e1/
kBEhIJKzQQy2axRUARwj12dDMFUsAYrk2QAwFA47ejkYoowAwB0gwk
OEVGRkMyRTAtQkQ1Qy00MTIGLTICQkMtMjRDRERBMzZBNDQw@WAA@=
```

This content data is then highlighted and copied. We saved it and decoded it using a base 64 decoder. In this case, we used James Eichbaum's Base64 Decoder (<http://github.com/eichbaumj/Python>). We then save the data as a .bin file and review it in the HxD Hex Editor (see Fig. 9.26). Protocol Buffer data does not have a file signature per se. If you recall, there are different Scalar values int32, int64, string, bytes, etc. . . . Well these all have associated wire types. We have to look for hex values like 08, 09, 0A. In a protobuf, those values all correspond to key = 1 with different wire types:

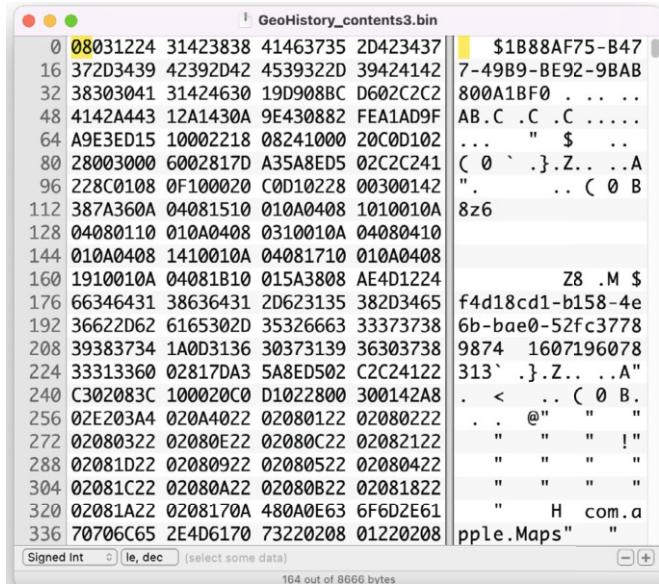


Fig. 9.26: Base64 Decode: GeoHistory Contents

08 varint (A variable length integer)
09 64 bit
0A length delimited

Table 9.6: Typical protobuf start values

A value like 08 is not a header byte. It is just a common value protobufs start with. If interpreted as key and type it translates to key = 1, type = varint. Another common byte at the beginning of protobufs is 0x0A which translates to key = 1, type = length delimited (i.e. nested message, string, byte array). Remember: We can only make an educated guess about the binary content since protobufs directly start with the serial data stream.

➤ Important

A file header for a GZIP File is Hex 1F8BC8 the file will then have to be saved and unzipped.

As we know now, a protocol buffer message is a series of key-value pairs. Therefore, the serialised message consists of a series of key-value pairs that are stored one after the other in the data stream. When a message is encoded, they are concatenated into a byte stream. The binary version of a protobuf message uses the field's number as the key. A concrete name and declared type for each field can only be determined on the decoding end by referencing the message type's definition (i.e. the `.proto` file). When the message is being decoded, the parser needs to skip fields that it does not recognise. This way, new fields can be added to a message without breaking old programs that do not know about them. To this end, the "key" for each pair in a wire-format message is two values – the field number from your `.proto` file, plus a wire type that provides just enough information to find the length of the following value. Mostly, this key is referred to as a tag [23]. Fig. 9.7 demonstrates the wire types available.

Type	Meaning	Usage
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-bit	fixed64, sfixed64, double
2	Length-delimited	string, bytes, embedded messages, packed repeated fields
3	Start group	groups (deprecated)
4	End group	groups (deprecated)
5	32-bit	fixed32, sfixed32, float

Table 9.7: Available Wire Types

Just recall the `formobilechat` message from the earlier section. The protobuf binary (small one in this case) is shown in Fig. 9.27. The figure demonstrates the raw data; notice that it starts with 0x08.

The problem is we can figure out where it starts with these Hex values, possibly, but where does it end? Is it the end of the file, or only for a few bytes? Or is it 188 bytes like the aforementioned example. That is when the developers reply, "Welcome to our world."

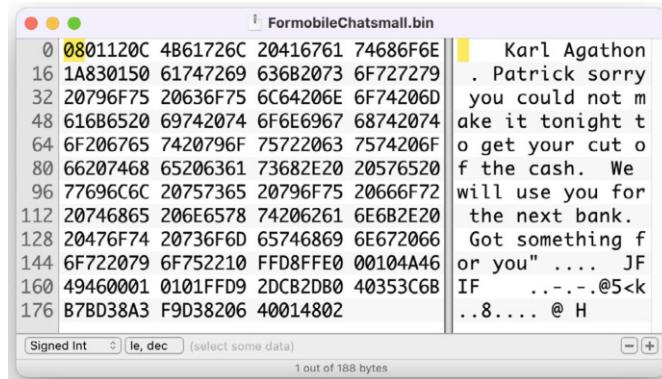


Fig. 9.27: Notice The Variable Integer 0x08 at the start

9.3.2 Yet another example: Apply Property List (PLIST) Files

Let's take a look at another example from our sample data. Fig. 9.28 shows data copied from the `GeoHistory.mapsdata.plist`. The value stored under the key "content" looks suspiciously like a BASE64 encoded value. And indeed, the data which appears to be BASE64 was copied and pasted into www.motobit.com decoder. The result was then copied and saved into notepad as `GeoHistory_contents3.bin` and opened into HxD.

Using the file `GeoHistory_contents3.bin` start with 08, but which one or ones? The file starts with 08. A search for Hex 08 results in 206 hits. See Fig. 9.29.

We manually look at each hit and the ASCII area for human-readable data that makes sense, which will not always be the case. In this first example, we take the results of the last hit, which starts at decimal offset 8579. Please copy the entire length to the end of the file and paste it into a new HxD file that we save and name with the offset (see Fig. 9.30).

Then the rest is decoding the data with Protoc and/or other tools. Moreover, try to figure out what we are looking at. Since the data belongs to a map application, we want to see if the hex values below are latitude and longitude. Maybe one of the other values is a date time stamp? See:

```
1: 0x4040cda6e5dc30e8
2: 0xc05c16c2f76aa800
```

Indeed, the values look like latitude and longitude in decimal. However, we do not want to assume that. This takes time and effort if we have to go through each search. Since the file started with hex 08, we can try protoc and our other tools against the entire `GeoHistory_contents3.bin` file. Keep in mind: Without the corresponding `.proto` file, we can only speculate about the meaning of the data.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>MSPHistory</key>
    <dict>
        <key>clientIdentifier</key>
        <string>0BB25E52-3840-48AE-90AF-DE804544E8E9</string>
        <key>records</key>
        <dict>
            <key>09E098EF-EDEC-4129-B539-551AB39CB9F1</key>
            <dict>
                <key>contents</key>
                <data>
                    CAE5JDA5RTA50EVGLUVERUMtNDEyOS1CNTM5LTU1MUFC
                    MzL0qjLGMRlihjz7AsLCOTJHChZQaG9lbuml4IGFyaXpv
                    bmEgYXJjYWRLegdBcmI6b25hIiOpAAQbMYWoQEAXAADY
                    82gYXMA5AAC4+yXZQEBBAAB2Mgj3W8BYAA==
                </data>
                <key>contentsTimestamp</key>
                <data>
                    C7JeijjhASK6Qr96ARUTo6QAAAAE=
                </data>
                <key>modificationDate</key>
                <date>2020-12-11T19:54:30Z</date>
            </dict>
            <key>17713B5A-4270-4CCE-BA34-398D761B5D1B</key>
            <dict>
                <key>modificationDate</key>
                <date>2020-12-10T22:00:20Z</date>
            </dict>
            <key>1B88AF75-B477-49B9-BE92-9BAB800A1BF0</key>
            <dict>
                <key>contents</key>
                <data>
                    CAMSJDFC00hBRjc1LU10NzctNDlC0S1CRTkyLTlC0UI4
                    MDBMUJGMbNzCLzWasLCQUKkQxKhQwqeQwiC/qGtn6nj
                    7RUQACIYCCQACDA0@QiaADAAAYAKBfNajtUCwsJB1owB
                    CA8QACDA0@QiaADABQjh6Ng0ECBUQAQ0eCBAQAQ0eCAEQ
                    A0eCAMOAQoECAQQAQoECBQAQoECBcQAQoECBkQAQoE
                    CBsQAVo4CK5NEiRmNGQxGNKMS1iMTU4LTRLnItymFl
                    MC01MmZjMzc3D0k4NzQaDTE2MDcxOTYwNzgzMNTngAoF9
                    o1q01QLCwKEiwwIIPBAAIMDRA1gAMAFCqAL1a60CCkA1
                </data>
            </dict>
        </dict>
    </dict>
</plist>

```

Fig. 9.28: GeoHistory.mapsdata.plist

9.3.3 Suggested Examination Process of a File

The idea is you create the process that works best for you. This may be completely working within your mobile forensic tool. The alternative is that you export out the file(s) of interest and review the data.

1. If unknown file type, then place the file into a Hex Viewer/Editor
2. Identify the File Signature (Research it if unknown to include possible file extensions). The data itself may be human readable as well.
3. Make sure the file has the right file extension
4. Open the file to view it natively i.e. as a database, xml, or plist file.
5. Look for Binary Large Objects (BLOB) or other raw data. Copy this raw data.
6. Decode this raw data with a Base64 Decoder and save. Devise a system to name the file and add a .bin file extension on it as a place holder.
7. Open this .bin file in a Hex Viewer/Editor
8. Identify the File Signature (as it could be an XML or Apple Property List file).
9. If this is an XML or PLIST File go back to Step 5. If not close the file and move on.

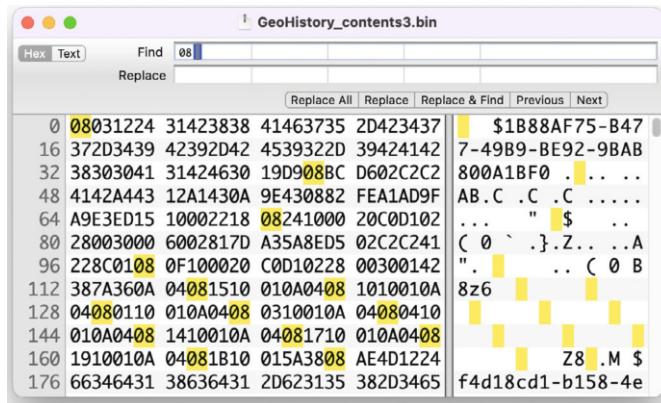


Fig. 9.29: Finding Protocol Buffer Data



Fig. 9.30: The extracted Protocol Buffer from offset 8579

```
Terminal — 74x14
$>protoc --decode_raw < GeoHistory_contents3_8579.bin
1: 1575007843489709826
2 {
  1: 0x4040cda6e5dc30e8
  2: 0xc05c16c2f76aa800
}
3: 9902
50: 1
8: 1377108822083
200: 1
202: "698C1449-9899-40FF-A3F1-B5482EDA28E9"
11: 0
$>
```

Fig. 9.31: Notice the Hex (0x) Values

10. Place the file in a folder with ProtoC executable.
11. Open a command prompt from this location
12. Type: `protoc --decode_raw < Filename.bin`
Click enter.



```
$>protobuf_inspector < GeoHistory_contents3_8579.bin
root:
  1 <varint> = 1575007843489709826
  2 <chunk> = message:
    1 <64bit> = 0x4040CDA6E5DC30E8 / 4629926533641548008 / 33.606656
    2 <64bit> = 0xC05C16C2F76AA800 / -4585765293939578880 / -112.35565
  3 <varint> = 9902
  50 <varint> = 1
  8 <varint> = 1377108822083
  200 <varint> = 1
  202 <chunk> = "69BC1449-9899-40FF-A3F1-B5482EDA28E9"
  11 <varint> = 0

$>
```

Fig. 9.32: Protobuf-Inspector converted Hex values. Lat and Long?

If you placed the file in the same directory as the *protoc* then to find the file automatically, without typing, after you type the “<” character click the space bar once and then hit tab to cycle through all of the files in the folder. Once you find the right file click enter.

13. See if you have Protocol Buffer data. If it Failed to parse the input, then that is not a protobuf (or you may need to review sections of the file for data).
14. To save the data. Highlight it and Left Click.
15. Paste into text editor. This allows you to use keywords as well.

! Attention

Now, remember the tools may obtain this data for you. However, it is nice to know where the data came from. Examining an unsupported application may have you uncovering protocol buffers for data as well.

9.3.4 Tools

We need to consider some of the tools in your toolbox for examining these artefacts. Some of these capabilities may be included with your mobile device forensic tools, such as MSAB’s XRY and XAMN. If so, then these non-forensic tools will help validate your work. Most are free or have a freeware version. Some of these may cost money, so look for Freeware versions:

- Hex Viewer/Editor: [HxD](#)
- Sqlite database viewer: [SQLite Expert](#)
- XML file viewer: [Notepad++](#)
- PLIST file viewer: [PLIST Editor for Windows](#)

- Base64 Decoder: [Motobit](#) and [James Eichbaum's Base64 Decoder.pyw](#)
- Epoch Timestamp Converter: [Epoch Converter Website](#) and [James Eichbaum's Tempus.pyw](#)
- File Signature Analysis: [Gary Kessler Website](#)
- Windows Calculator in Programmer Mode
- [Visual Studio](#)
- Protocol Buffer Compiler: [Proto C \(protoc.exe\)](#) and [Protobuf inspector](#)

9.4 Conclusion

We have seen why Protocol Buffers are helpful. They take data make it small, and provide faster transmission speed. Coders themselves may not want to welcome the structure. As forensic examiners, we learned that understanding this structured serializing data is essential. Applications use Protocol Buffers to store data in Apple Property Lists (Plists), Binary Large Objects, and XML Files. We may find user data, time stamps, location data, and more. So, these applications alone show that protocol buffers are used and the importance of understanding them and how to analyze them.

The most important takeaway that we can provide is that now you will hopefully identify what you are looking at. Have a greater appreciation of Protocol Buffers how to make sense of this data and explain it if necessary. We learned what to look for, and now you do also.

Acknowledgements My thanks to my fellow MSAB colleagues Johan Persson, Sebastian Zankl, Oscar Choi, and Global Training Manager James Eichbaum for their time, contributions to this chapter and the forensic community.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

