Automatic Schema Inference from Unknown Protobuf Messages

Jared Chandler
School of Electrical and
Computer Engineering
Dartmouth College
Hanover, New Hampshire
Email: jared.d.chandler@dartmouth.edu

***Abstract*—Packed binary formats present a challenge for network analysts, reverse engineers, and security researchers. Determining which fields are required, which fields are optional, and which fields can be repeated demands attention to detail, specialized human expertise, and ample time. We present an automatic schema inference approach targeting the widely used protobuf serialization format. Our approach recursively identifies field arity constraints from a collection of raw binary messages and reports them to a user as a complete protobuf schema. In our evaluation, this approach demonstrates high accuracy across a variety of inputs including real-world protobuf messages and binary files.**

1. Introduction

Protocol reverse engineering is a critical cyber-security task performed by researchers, security analysts, and network administrators [1], [2]. Unknown message formats present a significant obstacle, taking time and effort to manually reverse engineer [3]. Packed binary formats are particularly difficult for humans to reason about as the binary data is inherently ambiguous [4].

Protocol buffers or Protobuf[1] is one such packed binary message format widely used in enterprise software, mobile applications [5], and Internet of Things (IoT) devices [6], [7]. Protobuf is a self-describing format, where the individual fields of a message are serialized with a field datatype, and a field number.[2] This information is sufficient for the receiver to validate and interpret the message according to a specific schema. Unfortunately, the complete schema information is not contained in the message itself, and at present requires a human expert to infer it painstakingly by hand.

This paper focuses on recovering key schema structural information from the observation of multiple raw protobuf messages. Specifically, whether a field is optional, required, or can be repeated within a message. By observing the presence or absence of each field across a collection of messages, as well as the number of times each field is repeated in a single message, our approach infers a precise schema automatically, even in complex situations.

Our paper is organized as follows. First, we present a brief background on protocol reverse engineering and packed binary formats in Section 2. Next, we present our three main contributions:

- An algorithm for inferring protobuf schemas from raw binary messages in Section 3;
- An implementation of this algorithm as an end-user tool called FUBOTORP;
- An evaluation of our algorithm on example data in Section 4 including results from four real-world protobuf datasets.

We then discuss related works in Section 5 before concluding with an examination of the current limitations and avenues for future work in Section 6.

2. Background & Assumptions

Broadly, protocol reverse engineering is the process of recovering a precise specification for an unknown or undocumented protocol. Different approaches leverage different inputs including binary programs, traces of execution, source code, or in the most challenging cases only passively collected network traces. Protocol reverse engineering from traces for packed binary protocols is particularly challenging as these are largely devoid of meaningful text to aid an analyst. Instead, packed binary formats focus on efficient use of bandwidth, and flexibility across a variety of use cases. Serialization libraries and their accompanying serialized messages allow developers to avoid having to write binary message serialization and deserialization from scratch. Instead, by writing a high-level *schema* or description, the library produces this code automatically. This pattern is widespread and supported by several different tools including protobuf, flatbuffers[3], capnproto[4], and messagepack[5] to name a few. Among these protobuf one of the most widely

1. https://protobuf.dev/
2. https://protobuf.dev/programming-guides/encoding/
3. https://flatbuffers.dev
4. https://capnproto.org
5. https://msgpack.org

```
message Example {
   required int32  field1 = 1;
   repeated string field2 = 2;
   optional int32  field3 = 3;
   }
```

Figure 1. Example protobuf schema with three fields: a required 32-bit integer, a variable number of strings, and an optional 32-bit integer. Each field is defined with its constraints (required, repeated, optional), its data-type, an identifier (field1 $\cdots$ field3), and a field number ($1 \cdots 3$)

used across a variety of contexts []. The key feature of tools like protobuf is the ease of creating and using complex binary message formats. While this is great for a developer, it presents significant challenges for a reverse engineer, as the resulting messages vary widely in on-the-wire layout, number of fields, and field data-types. Packed binary formats handle this complexity by making the data self-describing. That is to say, by automatically including small pieces of meta-data to aid the deserializer. This information is library specific, and like all binary data, assumes some preshared agreement on how to interpret the data. As a result, reverse engineering packed binary formats such as protobuf can leverage these binary clues as a steppingstone to recovering a complete schema for the protocol.

Our approach to recovering these schemas is subject to the following three assumptions.

1) We have multiple raw messages which adhere to a single protobuf schema.
2) The messages are unencrypted and error-free.
3) We assume that each schema field appears at least once in the collection.

Multiple messages can be collected from a system or IoT device using network security tools such as burpsuite[6], mitm-proxy[7], or Wireshark[8]. These tools also include capabilities for capturing message clear-text through a Man-in-the-middle attack, use of a root-certificate, or simply exploiting the lack of certificate validation which is common in many IoT devices [8], [9], [10].

One scenario these assumptions cover is that of a security analyst who wants to understand the message format an IoT device uses. Understanding the format, and the semantics of the messages aids in identifying potential vulnerabilities for exploit or for further exploration via fuzzing [11], [12], [13]. Another scenario would be when a software engineer wants to extend a proprietary system with custom middleware but needs to develop a parser [14]. A third scenario would be a network administrator who wants to develop firewall rules to distinguish between innocuous and malicious network traffic [15], [16] or create traffic for testing network security tools [17].

We show an example protobuf schema in Figure 1, with the consuming automata in Figure 2 and an illustration of

6. https://portswigger.net/burp
7. https://mitmproxy.org
8. https://wireshark.org

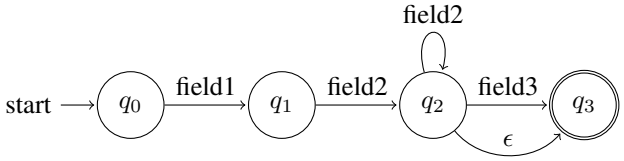

Figure 2. State-machine diagram of Example schema.

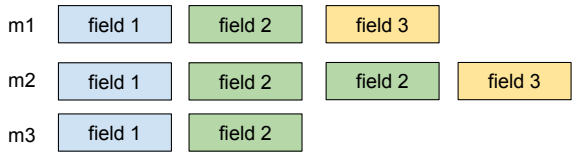

Figure 3. Illustration of three messages (m1 to m3) adhering to our example schema from Figure 1.

three example messages in Figure 3. This protobuf schema describes a message composed of three fields. Each field is defined with four pieces of information. First, a constraint on the number of times a field is present in a message. Second, a data-type for the field. Third, a field identifier. Fourth, a field number, which uniquely identifies the field in the message. One subtlety of protobuf is that while field numbers must all be positive integers, they do not need to be used sequentially, allowing for skipped numbers and gaps. When a protobuf message is received, the field number is used in combination with the data-type to interpret the field value and to check that the schema constraints on field repetition are satisfied. We now turn to how we use these field numbers to infer a schema from a collection of unknown protobuf messages.

3. Inference Method

Our goal is to recover a schema which describes a *collection* of observed protobuf messages. As individual protobuf fields are self-describing, the key obstacle is to recover whether a specific field is always present, if it is optional, and whether it ever appears multiple times. We call the field presence and repetition information the *arity constraints* of the field.

This information is expressed in the protobuf schema as the three keywords `required`, `optional`, and `repeated`. These arity keywords are the key pieces of

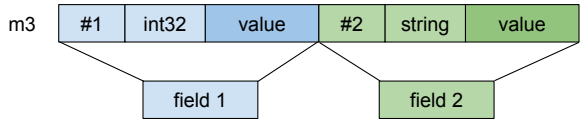

Figure 4. Illustration of on-the-wire layout of message m3. On the wire, each field has three sub-regions: the field number, the field type, and the field value or payload. For cases where the field is variable size, the field type also encodes information about the overall length of the payload.

information which we need to infer a schema consistent with the collection of messages. The other schema values–field data-types, field numbers–are contained in the messages themselves and can be directly transcribed to the inferred schema.

To recover the arity, we first interpret each message according to the protobuf primitives producing a list of fields numbers, and types. We then consider how many times each field occurs in each message. Then, by observing the presence or absence of a field across a collection of multiple messages we can determine if it is always present, if it is sometimes present, or if it is present more than once in a single message. We use these observations to infer the arity constraints of each field in the collection.

3.1. From Messages to Fields

We interpret each message according to the protobuf wire format specification. Each message consists of one or more *fields*. A *field* is composed of three pieces of information: the field *number*, the field *data-type*, and the field *value*. We illustrate an example of these field sections in Figure 4. Our focus here is on the field number as this uniquely identifies the field in the schema, and in the observed data. From our example, message m1 becomes the list of field numbers [1,2,3], message m2 the list [1,2,2,3], and m3 the list [1,2].

3.2. From Fields to Occurrences

For all distinct fields observed over a collection of messages, we create a *occurrence vector* for each field using the field number. The occurrence vector is defined as the number of times a field number is observed in each message of a collection. We define the *occurrence matrix* for a collection as the matrix comprised of the individual occurrence vectors and give an algorithm for calculating it in Algorithm 1.

Algorithm 1: Calculating Occurrence Matrix

```
Data: MSGS (Observed Messages)
Result: X (Occurrence Matrix)
FIELDS ← {fieldnum ∈ MSGS};
X ← Matrix |FIELDS| columns x |MSGS| rows;
for fieldnum ∈ FIELDS do
    for msg_i ∈ MSGS do
        X_{fieldnum,i} ← count of fieldnum in msg_i;
    end
end
```

Having calculated our occurrence vectors, we now turn to using them to infer the field arity constraints.

3.3. Interpreting Occurrence Vectors

Intuitively, the field arity constraints can be thought of as observed properties over the number of occurrences of a field. For example, when there are 0 occurrences of a field in some message in a collection, we can infer that the field must be optional in the schema. Similarly, when we observe that there are multiple occurrences of a field in some message, we can infer that the field must be repeatable according to the schema.

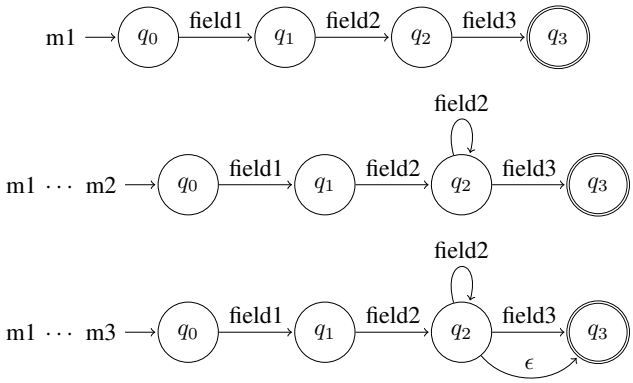

Figure 5. Sequential DFA construction from messages m1 through m3

We map our field number observations to arity constraints for use in the schema using the following equations.

We define two formal tests: OPTIONAL and REPEATED, along with one heuristic test: REQUIRED. The OPTIONAL test (Equation 1) determines if there is any instance where a field was not present in a message. Similarly, the REPEATED test (Equation 2) determines if there is any instance where a field appeared more than once in a single message. We consider these tests formal, as once they are satisfied by a collection of messages, no additional message can refute or negate them, and the underlying schema *must* have these constraints for the respective fields.

$$\text{OPTIONAL}(\mathbf{X}_{fieldnum}) = \exists x \in \mathbf{X}_{fieldnum}, x = 0 \quad (1)$$

$$\text{REPEATED}(\mathbf{X}_{fieldnum}) = \exists x \in \mathbf{X}_{fieldnum}, x > 1 \quad (2)$$

$$\text{REQUIRED}(\mathbf{X}_{fieldnum}) = \forall x \in \mathbf{X}_{fieldnum}, x \neq 0 \quad (3)$$

However, our heuristic test REQUIRED (Equation 3) *can* be negated by some additional message which omits the field, changing the arity in the schema from `required` to `optional`. We discuss the impact of additional messages in Section 3.4. Put simply, once a field is `optional` or `repeated` it will remain so, while a `required` field only holds for the data so far.

Intuitively, evaluating a collection of messages according to these tests can be likened to the process of refining a state machine for deserializing a message. In this process the OPTIONAL test adds ϵ transitions between states, while the REPEATED test adds self-loops. We illustrate this concept in Figure 5, showing the construction of the DFA for our example at each stage as new messages are evaluated.

Message	Field1	Field2	Field3
m1	1	1	1
m2	1	2	1
m3	1	1	0

TABLE 1. OCCURRENCE MATRIX FOR OBSERVED COUNTS OF THREE FIELDS FOR A SERIES OF THREE MESSAGES (M1 THROUGH M3).

```
message [_______] {
   required int32  [______] = 1;
   repeated string [______] = 2;
   optional int32  [______] = 3;
   }
```

Figure 6. Resulting inferred protobuf schema.

Returning to our example messages in Figure 3, we infer our arity constraints by using Algorithm 1 to construct an occurrence matrix as shown in Table 1. Once constructed, we then evaluate each occurrence vector according to OPTIONAL,REPEATED, and REQUIRED texts to infer the arity constraints for each field. We show the resulting inferred schema in Figure 6.

A note on protobuf schema syntax limits. While our approach can infer precise arity constraints, protobuf schema syntax cannot express these fully. First, the `repeated` keyword implicitly means any number of repetitions, including 0. As a result, `repeated` cannot be combined with the `required` keyword. The complete inferred arity constraints would likely be useful to an end-user and can easily be included in the resulting schema as annotations.

Ultimately the inferred schema is only representative of the observed data. If a field is present in every message observed, it may in fact be `required`, or we may have yet to see a message with a counterexample. We examine exactly this situation in the following section and discuss it further in our evaluation (Section 4).

3.4. Additional Messages

Next, we consider what happens to the arity constraints when we include a new message (m4) in our collection. We illustrate the expanded collection in Figure 7. Message m4 omits field 2 entirely, while including 3 instances of field 3. This message is at odds with the schema in Figure 6, and the final DFA in Figure 5. This scenario is consistent

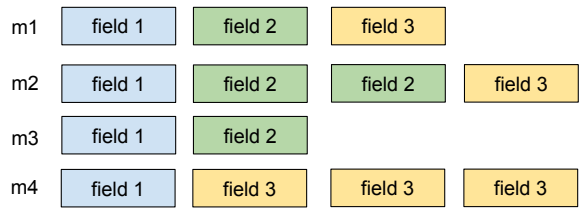

Figure 7. Illustration of four messages.

Message	Field1	Field2	Field3
m1	1	1	1
Inferred Constraints 1	Required	Required	Required
m2	1	2	1
Inferred Constraints 2	Required	Required Repeated	Required
m3	1	1	0
Inferred Constraints 3	Required	Required Repeated	Optional
m4	1	0	3
Inferred Constraints 4	Required	Optional Repeated	Optional Repeated

TABLE 2. OBSERVED ARITY CONSTRAINTS OF THREE FIELDS FOR A SERIES OF FOUR MESSAGES (M1 THROUGH M4).

with situations where a manufacturer or developer changes a protocol, to make an existing field optional or to add an entirely new field. A researcher must update their inferred schema to incorporate the newly observed message. Our inference algorithm can be applied one message at a time to capture how the arity constraints change. We illustrate this sequential application in Table 2.

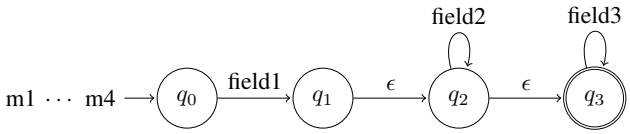

Figure 8. DFA construction from observed messages m1 through m4

```
message [_______] {
   required int32  [______] = 1;
   repeated string [______] = 2;
   repeated int32  [______] = 3;
   }
```

Figure 9. Example protobuf schema with three fields: a required 32-bit integer, a variable number of strings, and a variable number of 32-bit integers.

This table illustrates how our approach not only can produce a revised schema (Figure 9 and 8), but can help a researcher understand specific differences between groups of messages, such as those observed between two different versions of an IoT device.

3.5. Recursive Schema Inference

A key design feature of protobuf is the ability to define subformats as field datatypes, as shown in Figure 10. Subformat field values are serialized as complete protobuf messages according to the subformat schema. These subformat

```
message Format {
   required int32     field1 = 1;
   optional string    field2 = 2;
   repeated Subformat field3 = 3;
   }

message SubFormat {
   required string    field1 = 1;
   required int32     field2 = 2;
   }
```

Figure 10. Example protobuf schema with nested subformat.

bytestrings are inserted into the encapsulating message at the appropriate field position. Subformats allows schema authors to group fields according to semantics or intended use. Importantly, the field names and numbers of each subformat are independent of those from the encapsulating format and any other subformats. As the subformat bytestring values are independent of the encapsulating format, and satisfy the same assumptions, performing recursive format inference is trivially simple. First, when a field's on-the-wire serialization type is a bytestring, after inferring the arity as normal our algorithm recurses on the field bytestring values as a complete set of messages. If a schema is inferred by our algorthim from those messages, we update the field type with the subformat, leaving the field type as a bytestring otherwise. This simple extension allows our approach to recover complex nested schemas as we discuss next in our evaluation.

4. Evaluation

We evaluate the effectiveness of our approach on the task of automatic inference of schemas using only raw messages as input. As protobuf fields are encoded with a field number and type (Figure 4), our evaluation focuses on accurately inferring the *arity* of the fields, both for top-level formats, *and* nested subformats.

We conduct this evaluation in third phases. First, we evaluate our arity inference under controlled conditions on synthetic data (Section 4.2). Next, we evaluate FUBOTORP on 4 real-world datasets in scenarios consistent with a security analyst reverse engineering unknown file or network data formats (Section 4.3). Finally, we include a short case study detailing the effectiveness of our approach on a real-world unknown file (Section 4.4).

Our synthetic evaluation shows FUBOTORP correctly inferred the arity across most of our test cases, and perfectly for collections of 10 messages or more, a strong positive result. Our real-world evaluation shows FUBOTORP correctly inferred field arity in all four datasets subject to data diversity. Further, FUBOTORP's ability to infer arity recursively on nested subformats allowed it to produce schemas consistent with the ground truth hierarchy in all cases.

4.1. FUBOTORP Implementation

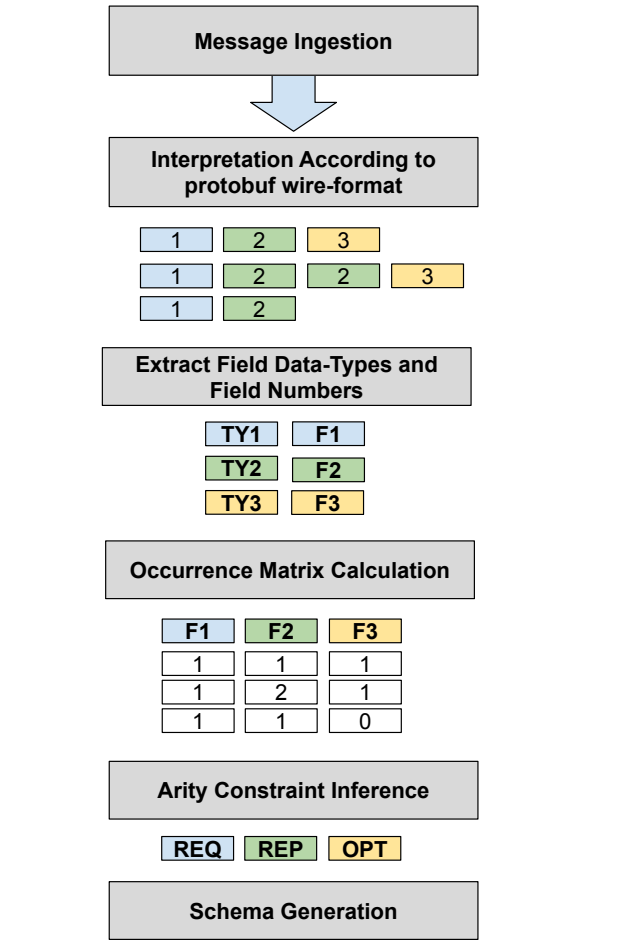

Figure 11. Illustration of the FUBOTORP inference steps.

To perform our evaluation, we implemented our algorithm as a python tool we call FUBOTORP. FUBOTORP takes as input a sequence of raw protobuf messages, and produces as output a protobuf version 2.0 schema. We illustrate the inference steps of our tool in Figure 11. Our approach is modular, allowing different on-the-wire interpretation approaches to be substituted. FUBOTORP creates placeholder field names, as these are not transmitted with the message, nor can the original names be easily inferred from the data-types alone as others have shown [18], [19]. FUBOTORP returns the inferred schema as ASCII text directly to the user. Our tool uses the protobuf wire-format data-type identifiers directly in the output schema as shown in Figure 11.

Field Layout	Number of Messages per Collection												
	1	2	3	4	5	6	7	8	9	10	15	20	25
required	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
optional	0.48	0.80	0.88	0.94	0.96	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
repeated	0.91	0.97	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Average (1 field)	0.80	0.92	0.96	0.98	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
required,required	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
optional,optional	0.52	0.72	0.84	0.97	0.96	0.97	0.99	0.98	1.00	1.00	1.00	1.00	1.00
repeated,repeated	0.89	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
required,optional	0.74	0.91	0.93	0.95	0.97	0.99	1.00	1.00	0.99	1.00	1.00	1.00	1.00
optional,required	0.50	0.75	0.85	0.94	0.99	0.99	0.99	0.99	1.00	1.00	1.00	1.00	1.00
required,repeated	0.95	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
repeated,required	0.87	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
optional,repeated	0.49	0.78	0.89	0.94	0.95	0.98	0.98	0.99	1.00	1.00	1.00	1.00	1.00
repeated,optional	0.71	0.90	0.94	0.95	0.99	0.99	0.98	1.00	0.99	1.00	1.00	1.00	1.00
Average (2 fields)	0.74	0.89	0.94	0.97	0.98	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00
required,required,required	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
optional,optional,optional	0.52	0.80	0.90	0.93	0.98	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
repeated,repeated,repeated	0.89	0.97	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
required,optional,repeated	0.68	0.81	0.91	0.97	0.98	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00
required,repeated,optional	0.82	0.90	0.95	0.98	1.00	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
optional,required,repeated	0.53	0.73	0.88	0.97	0.99	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
optional,repeated,required	0.46	0.77	0.86	0.95	0.97	0.99	0.99	0.99	0.99	1.00	1.00	1.00	1.00
repeated,required,optional	0.78	0.92	0.97	0.98	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
repeated,optional,required	0.62	0.81	0.92	0.95	0.97	0.99	0.99	0.99	1.00	1.00	1.00	1.00	1.00
Average (3 fields)	0.70	0.86	0.93	0.97	0.99	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00
Average (5 fields)	0.64	0.80	0.89	0.94	0.97	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00
Average (10 fields)	0.61	0.75	0.86	0.92	0.97	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00
Average (20 fields)	0.54	0.64	0.78	0.88	0.93	0.96	0.98	0.99	1.00	1.00	1.00	1.00	1.00

TABLE 3. EXPERIMENTAL RESULTS REPORTING THE AVERAGE ACCURACY PER FIELD LAYOUT AND NUMBER OF MESSAGES.

We plan to release FUBOTORP as open-source in the near future.

4.2. Synthetic Data Inference Evaluation

Our approach assumes representative data, that is to say, a collection of messages where optional fields are omitted in at least one message, and repeated fields are repeated in at least one message. When these assumptions are met our inference approach will have perfect accuracy, as there is no ambiguity in the data. However, in practice a researcher must work with the data they have, often in smaller quantities than they would like. To characterize the accuracy of our approach across a variety of conditions we generated collections of messages with varying numbers of fields. Optional fields were included in messages with a probability of 0.50, while repeated fields were generated uniformly with 0 to 9 repetitions per message. A repeated field thus had a 0.10 probability of being omitted from a message. For other parameter choices our results would differ.

We generated layouts of 1, 2, 3, 5, 10, and 20 fields. For layouts with 3 or less fields, we included all permutations across the three field arities (optional, repeated, required), as well as cases where all fields were of the same arity. For each layout we generated 100 collections with increasing numbers of messages. We included collections with only a single message but note that in real-world applications we would assume there are two or more messages for inference. For each test case we compared the ground-truth arity with the inferred arity. We consider the inferred arity keyword accurate if it matched the ground-truth arity, and incorrect otherwise. We report these results in Table 3.

Unsurprisingly, our approach displayed perfect accuracy across all test inputs using only a collection of 10 messages. We interpret this as a strong positive result. For collections with fewer than 10 messages, the most common error was for an optional field to be mistakenly inferred as having required arity for collections where the field was present in every message. Similarly, but less frequently, repeated fields which were either omitted, or appeared only a single time across a collection of messages were mistakenly inferred as optional or required. Again, these results characterize how our approach would perform in real-world situations where the distribution of field arities across a collection of messages varies by protocol use case. These results on synthetic data informed our observations on real-world datasets which we discuss next.

4.3. Real-World Protobuf Format Evaluation

We evaluate FUBOTORP on real-world protobuf formats, collected in the wild. We identified four datasets of public

```
message glyphs {
    repeated fontstack stacks = 1;
}

message fontstack {
    required string name = 1;
    required string range = 2;
    repeated glyph glyphs = 3;
}

message glyph {
    required uint32 id = 1;
    optional bytes bitmap = 2;
    required uint32 width = 3;
    required uint32 height = 4;
    required sint32 left = 5;
    required sint32 top = 6;
    required uint32 advance = 7;
}
```

```
message  Format {
  required Format_A field1 = 1;
}

message  Format_A {
  required bytes    field1 = 1;
  required bytes    field2 = 2;
  repeated Format_B field3 = 3;
}

message  Format_B {
  required int32    field1 = 1;
  optional bytes    field2 = 2;
  required int32    field3 = 3;
  required int32    field4 = 4;
  required int32    field5 = 5;
  required int32    field6 = 6;
  required int32    field7 = 7;
}
```

Figure 12. Comparison of ground-truth (left) and inferred (right) schemas for GLYPH dataset. Arities for 'repeated' and 'optional' fields are color coded.

protobuf data. These four datasets consist of two instances of binary file formats (VECTOR, GLYPH), and two instances of network traffic respectively (GRPC, MTA).

The GLYPH dataset is composed of three files serializing individual font characters or glyphs for use in map labeling, ranging from 10Kb to 129Kb in size.[9] The VECTOR dataset consists of five files serializing map shapes such as outlines of buildings, regional boundaries, and natural features.[10] These files ranged from 52Kb to 358Kb in size. The GRPC dataset consists of two messages–66 Bytes, and 179 bytes respectively–exchanged between a gRPC client and server.[11] gRPC leverages protobuf encoding for data exchange. Finally, the MTA dataset consists of six gRPC-web messages[12] from the Metro Transit Authority[13] (MTA) public datafeed API endpoint. These messages communicate the real-time status of MTA vehicles. Similar to gRPC, gRPC-web serializes data using protobuf and transports it over an HTTP connection. We collected the dataset messages from two different subway lines at one-minute intervals with message sizes ranging from 24Kb to 128Kb.

For each dataset, we used either a reference schema provided with the data (GLYPH, VECTOR, GRPC), or the standardized schema referenced by the API (MTA). All schemas included nested subformats, making them excellent tests of FUBOTORP's ability to infer arity given complex schema hierarchies.

Next we validated that these reference schemas parsed the data using the protobuf compiler command `protoc`. [14] The only deviations reported by `protoc` were instances where some `fontstack` subformats in the GLYPH dataset omitted the `range` field. This divergence highlights an important motivation for protobuf reverse engineering: out-of-band verification of data formats. Protobuf's on-the-wire format is designed to allow a deserializer to skip over fields and subformats at will. This design decision means that data divergence may only be discovered if and when a field is deserialized. An advantage of this choice is that a complete schema is not needed to deserialize a subset of fields, only a schema which is complete relative to that subset. We elected to use these datasets, divergences included as these are the conditions under which real-world reverse engineering takes place.

While protobuf encodes datatypes in the on-the-wire serialization, the serialized datatypes are sometimes less precise than schema datatypes. For example, both strings and byte strings are serialized as a `bytes` field, requiring the schema to tell a parser when to differentiate. Similarly, a variety of integer fields are serialized as variable length integers, regardless of whether the values are signed, unsigned, or represent a small range of values such as an enumeration. FUBOTORP reports the schema equivalent of the on-the-wire serialized type. For strings, this means FUBOTORP reports a type of bytes. For varint (variable length binary encoded integers) numeric formats such as signed and unsigned 32-bit integers, FUBOTORP reports the `int32` type. Again, the focus of our evaluation is field arity, especially the field arity of nested subformats. While several tools have been developed for deserializing the on-

9. https://github.com/mapbox/glyph-pbf-composite
10. https://github.com/klokantech/mapbox-gl-js-offline-example
11. https://grpc.io/blog/wireshark
12. https://api.mta.info/#/subwayRealTimeFeeds
13. https://www.mta.info/
14. https://protobuf.dev/installation

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
  repeated PhoneNumber phone = 4;
  optional Timestamp last_updated = 5;
  optional bytes portrait_image = 6;
}

message PhoneNumber {
  required string number = 1;
  optional PhoneType type = 2;
}
```

```
message  Format {
  required bytes    field1 = 1;
  required int32    field2 = 2;
  required bytes    field3 = 3;
  repeated Format_A field4 = 4;
  optional Format_B field5 = 5;
  optional bytes    field6 = 6;
}

message Format_A {
  required bytes    field1 = 1;
  optional int32    field2 = 2;
}

message Format_B {
  required int32    field1 = 1;
}
```

Figure 13. Comparison of ground-truth (left) and inferred (right) schemas for the GRPC dataset. Arities for 'repeated' and 'optional' fields are color coded. In the ground-truth schema, the definition of Timestamp message format as a 32bit integer is pulled from an external reference.

the-wire encodings[15][16][17], some employing simple heuristics to make intelligent choices between bytestreams and strings, we focus on overall format structure and leave improved datatype inference for future work.

For each dataset, we ran FUBOTORP with the binary files as input and captured the inferred schema. We then used the inferred schema to parse input file validating that our inferred schema was consistent with the data. For all four datasets our inferred schemas were consistent with the data and reported no parsing errors. We consider this a strong positive result showing the utility of our tool to infer arities and produce schemas under real-world conditions. We next examined the inferred schema for each dataset respectively. To aid this analysis, we ordered the inferred schema message formats consistent with the corresponding ground-truth schema message formats, starting with the root or outer most message format. We now discuss each of the four inferred schemas in detail.

4.3.1. GLYPH Dataset. We report our inferred schema for the GLYPH dataset compared to the ground-truth schema in Figure 12. The GLYPH dataset has two layers of nested submessage formats. FUBOTORP was able to both identify the subformats, and identify the arity of the fields within. As the dataset messages each included only a single `fontstack` submessage the inferred arity for this field (`Format field1`) was inferred only as `required`. This result illustrates that inference is limited to the diversity of the underlying dataset, reinforcing the our observations from Section 4.2. For `Format_A field3`, and `Format_B field2`, the respective arities of repeated and optional were correctly inferred, a strong positive result.

4.3.2. GRPC Dataset. We report our inferred GRPC dataset schema compared to GRPC ground-truth schema in Figure 13. The GRPC dataset has two layers of nested subformats. FUBOTORP correctly inferred the organization and arities of the fields, save `Format field3`, which was present in all messages and as a result inferred as `required` instead of `optional`. One subformat (`Timestamp`) was defined in the ground-truth schema as an import from an external schema corpus. As FUBOTORP uses only the data in the messages, it inferred the underlying representation of that subformat as a 32-bit field in `Format_B`.

4.3.3. VECTOR Dataset. The VECTOR dataset is composed of three layers of nested subformats. For the outer two layers FUBOTORP performed similarly to the GLYPH and GRPC datasets as shown in Figure 14. In the underlying data `Format_A field 5`, was present in all submessages and as a result was inferred to be required instead of optional. For the nested subformats `Format_B` and `Format_C`–Feature and Value in the ground truth–we observed that the on-the-wire datatypes did not match the ground-truth schema for several fields. For two fields this difference was due to the data using a packed (compressed) serialization. FUBOTORP's on-the-wire deserializer presently expects unpacked values. In several cases where the ground-truth schema specified a 64-bit value, FUBOTORP reported 32-bit type. This discrepancy is due to the use of varints (variable length binary encoded integers). Protobuf attempts

15. https://github.com/mildsunrise/protobuf-inspector
16. https://github.com/vpetrigo/rev-protobuf
17. https://github.com/pawitp/protobuf-decoder

```
message Tile {
  repeated Layer layers = 3;
}

message Layer {
  required string name = 1;
  repeated Feature features = 2;
  repeated string keys = 3;
  repeated Value values = 4;
  optional uint32 extent = 5;
  required uint32 version = 15;
}

message Feature {
  optional uint64 id = 1;
  repeated uint32 tags = 2 [packed=true];
  optional GeomType type = 3;
  repeated uint32 geometry = 4 [packed=true];
}

message Value {
  optional string string_value = 1;
  optional float float_value = 2;
  optional double double_value = 3;
  optional int64 int_value = 4;
  optional uint64 uint_value = 5;
  optional sint64 sint_value = 6;
  optional bool bool_value = 7;
}
```

```
message  Format {
  repeated Format_A field3 = 3;
}

  message  Format_A {
    required bytes    field1 = 1;
    repeated Format_B field2 = 2;
    repeated bytes    field3 = 3;
    repeated Format_C field4 = 4;
    required int32    field5 = 5;
    required int32    field15 = 15;
  }

  message  Format_B {
    required int32    field1 = 1;
    required bytes    field2 = 2;
    required int32    field3 = 3;
    required bytes    field4 = 4;
  }

  message  Format_C {
    optional bytes    field1 = 1;
    // No field 2
    optional int64    field3 = 3;
    optional int32    field4 = 4;
    // No field 5
    // No field 6
    // No field 7
  }
```

Figure 14. Comparison of ground-truth (left) and inferred (right) schemas for the VECTOR dataset. Arities for 'repeated' and 'optional' fields are color coded. Annotations added to aid the reader are prefixed with "//". The differences in the lower two pairs of message formats arise from use of packed values, and 32-bit varints being cast up to 64-bit types.

to make messages as short as possible by removing leading 0 bits from multibyte integer fields. When deserialized, if the schema integer type is larger than the value, the value is cast up. These nuances underscore the importance of comparing multiple messages or data instances for schema inference, as some properties will occur only on occasion, and not in every observed field value.

4.3.4. MTA Dataset. Our final dataset, MTA, was the most complex in terms of structure. The ground truth-schema defines 28 message formats, nested up to six-layers deep. Given the six binary messages as input, FUBOTORP recovered 22 message formats and the accompanying hierarchical structure from the data alone, a strong positive result considering the complexity of the data. We report our inferred MTA schema compared with the ground-truth schema for the outer three format levels in Figure 15. The inferred schema highlights the importance of obtaining a representative sample with respect to the overall schema. Specifically, in the MTA dataset, many of the ground-truth schema optional fields, and several subformats were never observed in the collected data. As a result the field numbers for inferred fields contained gaps. While there is no requirement that a schema designer use field numbers sequentially, or under any restriction other than avoiding duplicates, most schemas we observed do in fact number fields sequentially. This information is useful to a reverse engineer, indicating inferred schema gaps which additional messages might fill in. In Figure 15 we add annotations to indicate these gaps as an aid to the reader. The complexity of the inferred schema serves as a reminder that the outputs of reverse engineering automation require appropriate interfaces to support interpretation by human experts [4].

4.4. Case Study: Unknown file

We conclude our evaluation with short case study in format reverse-engineering drawn from the real world. We identified a public forum post where a user asked for help deserializing an 86 megabyte binary file with an unknown schema.[18][19] This post offered the opportunity to test FUBOTORP in a real-world scenario on a large real-

18. https://groups.google.com/g/protobuf/c/coqYvMbNURw/
19. https://drive.google.com/file/d/12sBfxPsG1s2rVSewnkr_2ptfq4102b5M

```
message FeedMessage {
  required FeedHeader header = 1;
  repeated FeedEntity entity = 2;
}

message FeedHeader {
  required string gtfs_realtime_ver = 1;
  optional Incr incr = 2;
  optional uint64 timestamp = 3;
  optional string feed_version = 4;
}

message FeedEntity {
  required string id = 1;
  optional bool is_deleted = 2;
  optional TripUpdate trip_update = 3;
  optional VehiclePosition vehicle = 4;
  optional Alert alert = 5;
  optional Shape shape = 6;
  optional Stop stop = 7;
  optional TripMods trip_mods = 8;
}

message TripUpdate {
  required TripDescriptor trip = 1;
  repeated StopTimeUpdate stop_time_update = 2;
  optional VehicleDescriptor vehicle = 3;
  optional uint64 timestamp = 4;
  optional int32 delay = 5;
  optional TripProperties trip_properties = 6;
}

message VehiclePosition {
  optional TripDescriptor trip = 1;
  optional Position position = 2;
  optional uint32 current_stop_sequence = 3;
  optional VehicleStopStatus current_status = 4;
  optional uint64 timestamp = 5;
  optional CongestionLevel congestion_level = 6;
  optional string stop_id = 7;
  optional VehicleDescriptor vehicle = 8;
  optional OccupancyStatus occupancy_status = 9;
  optional uint32 occupancy_percentage = 10;
  repeated CarriageDetails multi_carriage_dets = 11;

}

message Alert {
  repeated TimeRange active_period = 1;
  // No field 2
  // No Field 3
  // No field 4
  repeated EntitySelector informed_entity = 5;
  optional Cause cause = 6;
  optional Effect effect = 7;
  optional TranslatedString url = 8;
  // No Field 9
  optional TranslatedString header_text = 10;
  optional TranslatedString description_text = 11;
  optional TranslatedString tts_header_text = 12;
  optional TranslatedString tts_description_text = 13;
  optional SeverityLevel severity_level = 14;
  optional TranslatedImage image = 15;
  optional TranslatedString image_alternative_text = 16;
  optional TranslatedString cause_detail = 17;
  optional TranslatedString effect_detail = 18;
}
```

```
message  Format {  //FeedMessage
  required Format_A field1 = 1;
  repeated Format_E field2 = 2;
}

message  Format_A { //FeedHeader
  required bytes    field1 = 1;
  // No field 2
  required int32    field3 = 3;
  // No field 4
}

message  Format_E { //FeedEntity
  required bytes    field1 = 1;
  // No field 2
  optional Format_F field3 = 3;
  optional Format_M field4 = 4;
  optional Format_P field5 = 5;
  // No field 6
  // No field 7
  // No Field 8
}

message  Format_F { //TripUpdate
  required Format_G field1 = 1;
  repeated Format_I field2 = 2;
  // No field 3
  // No field 4
  // No Field 5
  // No field 6
}

message  Format_M { //VehiclePosition
  required Format_N field1 = 1;
  // No field 2
  optional int32    field3 = 3;
  optional int32    field4 = 4;
  required int32    field5 = 5;
  // No field 6
  required bytes    field7 = 7;
  // No field 8
  // No Field 9
  // No field 10
  // No Field 11
}

message  Format_P { //Alert
  // No field 1
  // No field 2
  // No Field 3
  // No field 4
  optional Format_Q field5 = 5;
  // No field 6
  // No field 7
  // No Field 8
  // No Field 9
  required Format_T field10 = 10;
  // No Field 11
  // No Field 12
  // No Field 13
  // No Field 14
  // No Field 15
  // No Field 16
  // No Field 17
  // No Field 18
}
```

Figure 15. Comparison of ground-truth (left) and inferred (right) schemas for the MTA dataset showing the top three levels of format hierarchy. Arities for 'repeated' and 'optional' fields are color coded. Annotations added to aid the reader are prefixed with "//".

```
message  Format {
  repeated Format_A field1 = 1;
}

message  Format_A {
  repeated bytes    field2 = 2;
  repeated int32    field4 = 4;
  repeated int32    field5 = 5;
  repeated int32    field8 = 8;
  repeated int32    field9 = 9;
  repeated int64    field10 = 10;
  repeated bytes    field11 = 11;
}
```

Figure 16. Inferred schema for unknown file. Arities for 'repeated' fields are color coded.

world file. We report the inferred schema for this file in Figure 16. While no ground-truth schema was provided, we examined the deserialized data and found the structure consistent with the inferred schema, save for one divergence. We observed that the data in `Format_A field 2` was almost always a string containing a domain name, but on occasion was a bytestring which could be interpreted as a protobuf subformat. We were unable to construct a valid protobuf schema consistent with this representation. As both strings and serialized protobuf messages are bytestrings, we concluded that this was a case where subformat bytestring and string values were intermingled in a single field. Custom deserialization logic could interpret the value of this field by first attempting to deserialize as a protobuf message, and should that fail, reverting to interpreting as a simple string. While FUBOTORP does not currently support this type of mixed inference, it could be added through future work.

5. Related Work

Our approach focuses on directly inferring a schema from protobuf messages. This work is most closely related to research aimed at automatic protocol reverse engineering. These efforts are well summarized by Narayan et. al and Kleber et. al [1], [2]. Broadly, protocol reverse engineering focuses on recovering a specification from some artifact of the original system, usually a network trace [20], [21], [22], [23], a binary executable [24], [25], [26], or a trace of the running program itself [27].

Research leveraging network traces as input can be categorized as producing either a precise semantic description of the protocol, or a heuristic description of the observed data. Examples of a precise description include an inferred grammar, or generated parser. Chandler et al.'s BinaryInferno and Pohl et al.'s AWRE are two examples of approaches which produce these precise semantic specifications [20], [21]. Alternatively, a heuristic description of observed data often equates to byte-strings being segmented into guessed fields as is employed by Netplier [22] and Bossert's Netzob [28]. These heuristic guesses require further work by an analyst to refine into something useful.

Other relevant related work draws on research into grammar inference of regular languages. Of these, Angulin's work on L* is directly related to techniques to infer state machines for unknown network protocols [29], [30]. A key take-away from Angulin's work is that both positive and negative examples are necessary to correctly learn a format. As we observed in our evaluation, correctly inferring optional and repeated arities requires observing the absence of a field, and the repetition respectively.

6. Limitations & Future Work

Our approach to inferring schemas from raw protobuf messages has four main limitations. First, our approach is limited to protobuf as it leverages the serialized field numbers contained in protobuf messages. For other general serialization frameworks such as flatbuffers, msgpack and capnproto, our approach could be adapted if those formats include similar information. One example would be to leverage the type field a portion of type-length-value encodings as a stand-in for field number. Second, our approach requires messages consistent with a single schema. Messages across different schemas could introduce conflicting arity constraints and types for fields. One avenue of future work would be to detect these conflicts for protobuf messages automatically as a basis for segmenting the dataset before performing inference. Third, the protobuf on-the-wire format has allowances for append-only or last-one-wins fields. In these situations, if a field appears multiple times in a message, only the last instance of the field is deserialized. This feature is primarily used for on-disk file formats, as network use would waste bandwidth. At present our approach would consider these instances as repetitions. Finally, our approach cannot recover detailed schema meta-data such as individual field names. This information is not transmitted with the message. One potential solution to this would be to use a combination of field data-type, and context to make a guess regarding a field name. Leveraging the actual distribution of field values to infer these names would be another avenue for future exploration.

Acknowledgments

This paper results from the SPLICE research program, supported by a collaborative award from the National Science Foundation (NSF) SaTC Frontiers program under award number 1955805. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of NSF. Any mention of specific companies or products does not imply any endorsement by the authors, by their employers, or by the NSF.

References

[1] S. Kleber, L. Maile, and F. Kargl, "Survey of protocol reverse engineering algorithms: Decomposition of tools for static traffic analysis," *IEEE Communications Surveys & Tutorials*, vol. 2018, 2018.

[2] J. Narayan, S. K. Shukla, and T. C. Clancy, "A survey of automatic protocol reverse engineering tools," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, pp. 1–26, 2015.

[3] G. Bossert and F. Guihery, "Security evaluation of communication protocols in common criteria," in *Proc of IEEE International Conference on Communications*, 2012.

[4] S. Katcher, J. Mattei, J. Chandler, and D. Votipka, "An investigation of interaction and information needs for protocol reverse engineering automation," in *To Appear: Proceedings of the 2025 CHI conference on human factors in computing systems*, 2025.

[5] C. Currier, "Protocol buffers," in *Mobile Forensics–The File Format Handbook: Common File Formats and File Systems Used in Mobile Devices*. Springer, 2022, pp. 223–260.

[6] S. Popić, D. Pezer, B. Mrazovac, and N. Teslić, "Performance evaluation of using protocol buffers in the internet of things communication," in *2016 international conference on smart systems and technologies (SST)*. IEEE, 2016, pp. 261–265.

[7] S. Popić, I. Papp *et al.*, "Processing cost in case of message parsing on the smart iot gateway: Exploring the costs of unifying the message format to protocol buffer," in *2017 International Conference on Smart Systems and Technologies (SST)*. IEEE, 2017, pp. 169–173.

[8] M. B. Barcena and C. Wueest, "Insecurity in the internet of things," *Security response, symantec*, vol. 20, 2015.

[9] Z. Cekerevac, Z. Dvorak, L. Prigoda, and P. Cekerevac, "Internet of things and the man-in-the-middle attacks–security and economic risks," *MEST Journal*, vol. 5, no. 2, pp. 15–25, 2017.

[10] L. De Carli, R. Torres, G. Modelo-Howard, A. Tongaonkar, and S. Jha, "Botnet Protocol Inference in the Presence of Encrypted Traffic," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[11] B. Blumbergs and R. Vaarandi, "Bbuzz: A bit-aware fuzzing framework for network protocol systematic reverse engineering and analysis," in *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*. IEEE, 2017, pp. 707–712.

[12] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2015, pp. 330–347.

[13] R. Shapiro, S. Bratus, E. Rogers, and S. Smith, "Identifying vulnerabilities in scada systems via fuzz-testing," in *Critical Infrastructure Protection V: 5th IFIP WG 11.10 International Conference on Critical Infrastructure Protection, ICCIP 2011, Hanover, NH, USA, March 23-25, 2011, Revised Selected Papers 5*. Springer, 2011, pp. 57–72.

[14] P. C. Johnson, S. Bratus, and S. W. Smith, "Protecting against malicious bits on the wire: Automatically generating a usb protocol parser for a production kernel," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, 2017, pp. 528–541.

[15] P. Anantharaman, K. Palani, R. Brantley, G. Brown, S. Bratus, and S. W. Smith, "Phasorsec: Protocol security filters for wide area measurement systems," in *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*. IEEE, 2018, pp. 1–6.

[16] J. Chandler, K. Fisher, E. Chapman, E. Davis, and A. Wick, "Invasion of the botnet snatchers: A case study in applied malware cyberdeception," in *Proceedings of the 53rd Hawaii International Conference on System Sciences*, 2020.

[17] J. Chandler and A. Wick, "Synthesizing intrusion detection system test data from open-source attack signatures," in *2023 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2023, pp. 198–208.

[18] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 38–49.

[19] R. Bavishi, M. Pradel, and K. Sen, "Context2name: A deep learning-based approach to infer natural variable names from usage contexts," *arXiv preprint arXiv:1809.05193*, 2018.

[20] J. Chandler, A. Wick, and K. Fisher, "Binaryinferno: A semantic-driven approach to field inference for binary message formats," in *Proceedings of the Symposium on Network and Distributed System Security (NDSS'23)*, 2023.

[21] J. Pohl and A. Noack, "Automatic wireless protocol reverse engineering," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019.

[22] Y. Ye, Z. Zhang, F. Wang, X. Zhang, and D. Xu, "NetPlier: probabilistic network protocol reverse engineering from message traces," in *Proceedings of the Symposium on Network and Distributed System Security (NDSS'21)*, 2021.

[23] S. Kleber, F. Kargl, M. State, and M. Hollick, "Network message field type clustering for reverse engineering of unknown binary protocols," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2022, pp. 80–87.

[24] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007, pp. 317–329.

[25] J. Lim, T. Reps, and B. Liblit, "Extracting output formats from executables," in *2006 13th Working Conference on Reverse Engineering*. IEEE, 2006, pp. 167–178.

[26] J. Newsome, D. Brumley, J. Franklin, and D. Song, "Replayer: Automatic protocol replay by binary analysis," in *Proceedings of the 13th ACM conference on Computer and communications security*, 2006, pp. 311–321.

[27] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution." in *NDSS*, vol. 8. Citeseer, 2008, pp. 1–15.

[28] G. Bossert, F. Guihéry, and G. Hiet, "Towards automated protocol reverse engineering using semantic information," in *Proceedings of the 9th ACM Aymposium on Information, Computer and Communications Security*, 2014, pp. 51–62.

[29] D. Angluin, "Inductive inference of formal languages from positive data," *Information and control*, vol. 45, no. 2, pp. 117–135, 1980.

[30] ——, "Learning regular sets from queries and counterexamples," *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987.