BinaryInferno: A Semantic-Driven Approach to Field Inference for Binary Message Formats

Jared Chandler
Tufts University
jared.chandler@tufts.edu

Adam Wick
Fastly
awick@fastly.com

Kathleen Fisher
DARPA
katheen.fisher@darpa.mil

***Abstract*—We present BinaryInferno, a fully automatic tool for reverse engineering binary message formats. Given a set of messages with the same format, the tool uses an ensemble of detectors to infer a collection of partial descriptions and then automatically integrates the partial descriptions into a semantically-meaningful description that can be used to parse future packets with the same format. As its ensemble, BinaryInferno uses a modular and extensible set of targeted detectors, including detectors for identifying atomic data types such as IEEE floats, timestamps, and integer length fields; for finding boundaries between adjacent fields using Shannon entropy; and for discovering variable-length sequences by searching for common serialization idioms. We evaluate BinaryInferno's performance on sets of packets drawn from 10 binary protocols. Our semantic-driven approach significantly decreases false positive rates and increases precision when compared to the previous state of the art. For top-level protocols we identify field boundaries with an average precision of 0.69, an average recall of 0.73, and an average false positive rate of 0.04, significantly outperforming five other state-of-the-art protocol reverse engineering tools on the same data sets: Awre (0.18, 0.03, 0.04), FieldHunter (0.68, 0.37, 0.01), Nemesys (0.31, 0.44, 0.11), Netplier (0.29, 0.75, 0.22), and Netzob (0.57, 0.42, 0.03). We believe our improvements in precision and false positive rates represent what our target user most wants: semantically meaningful descriptions with fewer false positives.**

I. Introduction

Reverse engineering message formats from static network traces is a difficult and time consuming security task [31], [39], [50], critical for a variety of purposes: bug-finding via fuzz testing, automatic exploit generation, understanding the communications of hostile systems, and recovering specifications that are proprietary or have been lost. In prior work, researchers have used message reverse engineering techniques to gain insight into the behavior of malware [15], [25], [28] and manipulate botnets during mitigation efforts [23].

Protocol reverse engineering is characterized by a pipeline with multiple steps. These steps include collecting data, clustering messages by format, inferring a state machine describing how messages are exchanged, and most critically: inferring semantics for each format. The ambiguous nature of binary data makes such reverse engineering difficult. The same sequence of four bytes could be interpreted as an integer, a float, a string, a timestamp, *etc.*, or even several smaller fields. The *field inference* problem involves inferring field boundaries and the corresponding semantics. Without methods to precisely and automatically solve the field inference problem, the task falls to human experts. The working assumption underlying this work is that a tool that accurately identifies even some field boundaries and infers meaningful semantics for the corresponding data would make the job of these human experts easier *as long as the tool was almost always right*, *i.e.*, it could be trusted not to lead the experts down false-positive rabbit holes.

Our work addresses the experts' need for a security tool focusing on inferring semantics for each format. BinaryInferno is designed to work automatically and produce output descriptions that can be used to parse both existing and future messages directly. BinaryInferno tackles the field inference problem by leveraging an ensemble of detectors tuned to common protocol engineering tasks. In this paper, we limit our attention to common portions of a data format, *i.e.* data in which some portion of every message contains the same fields in the same order, including variable-length fields (such as variable-length strings or a repeating pattern of structured data). We exclude formats that include union types, although even in such formats, we will be able to infer the shared portions of the message. We believe that solving this problem is useful in its own right, *e.g.*, for identifying the structure of the shared header of variable payloads, and it is an important building block in solving the more general problem.

The key insight of our approach is that an ensemble of specialized detectors can provide more information, with fewer false positives, than a single, complex inference engine. Each detector in our extensible ensemble is responsible for analyzing the input sample of messages and returning a partial description in a data description language, identifying if and where its kind of data is likely to appear in the sample. We have built a number of such detectors, including *atomic detectors* for identifying basic semantic datatypes such as IEEE 754 floats [14], timestamps, and length fields; a *field-boundary detector* that uses changes in entropy to identify where one field ends and another begins; and a *pattern-based detector* that uses iterative deepening search guided by common serialization patterns to find variable-length fields.

Once the individual detectors have returned their results, we then automatically integrate the partial descriptions into

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.23131
www.ndss-symposium.org

a single description that accurately describes as much of the sample as possible. The key challenge in integrating the results is resolving conflicts in which different detectors make conflicting claims about particular bytes in the format. We resolve such conflicts by maximizing the amount of data described. The resulting description can be used to parse the sample data and – to the extent that the inference was correct – any future data from the same source. We have implemented our approach in a tool that we will release with an open-source license called BINARYINFERNO.

Our approach offers four main advantages. *Flexibility*: Each detector can use whatever method makes the most sense for identifying its kind of data. Many of the methods require only small samples and little or no training data. *Separation of concerns*: Each detector can focus on its own task in isolation. This separation means detectors for identifying fixed-width fields can work independently from detectors focused on identifying variable-length fields typically found in payloads, rather than trying to do both at once. *Parallelizability*: The task of running all the detectors is embarrassingly parallel, so the system automatically runs them in parallel. *Generality*: Because the system returns a data description suitable for guiding a parser rather than simply a set of field boundaries, it can be used for parsing future instances of similar data, not simply understanding the current sample.

After providing a short primer on relevant background material (Section II) and introducing our running example (Section III), we present our work on BINARYINFERNO, which makes the following contributions:

- We introduce the idea of using a semantics-driven ensemble to solve the field inference problem (Section IV). Our approach is characterized by using small specialized programs to independently identify parts of messages with specific semantic types. This approach has the twin benefits of automatically producing a semantically meaningful description for the analyst and strongly avoiding false positives.
- We present a field-boundary detector that relies on Shannon entropy (Section VI). The advantage of this detector is that it can often differentiate fields without knowing their semantic content, identifying field boundaries for otherwise unknown field types.
- We describe a search-based approach to inferring common serialization patterns for variable-length data (Section VII). Inferring the format of variable-length fields is one of the most challenging aspects of protocol reverse engineering. Our approach fits combinations of serialization patterns to the data, producing precise, semantically-meaningful descriptions.
- We introduce an integration algorithm for reconciling a collection of partial descriptions to find the best overall description (Section VIII).
- We evaluate BINARYINFERNO on 36 problem instances drawn from 10 extant binary protocols (Section IX). and compare BINARYINFERNO with five state-of-the-art protocol reverse engineering tools: AWRE [53], FIELDHUNTER [17], NEMESYS [38], NETPLIER [64], and NETZOB [20]. BINARYINFERNO identifies field boundaries in top-level protocols with an average precision of 0.69, an average recall of 0.73, and a false positive rate (FPR) of 0.04; our results mark significant improvements in precision and false positive rate compared to AWRE (0.18, 0.03, 0.04), FIELDHUNTER (0.68, 0.37, 0.01), NEMESYS (0.31, 0.44, 0.11), NETPLIER (0.29, 0.75, 0.22), and NETZOB (0.57, 0.42, 0.03) on the same data sets. These results reflect our general goal of finding true positives with high confidence while strongly avoiding false positives.
- We evaluate false positive rates for BINARYINFERNO, and related tools on uniform random data, which has no field boundaries; our results show that BINARYINFERNO rejects random data perfectly, as opposed to other tools which infer spurious boundaries at rates of up to 0.45.

II. BACKGROUND AND ASSUMPTIONS

There are three common approaches to field inference from network traces: heuristic approaches, sequence alignment techniques, and semantic approaches. Heuristic approaches, such as used in NEMESYS [38], identify fields using characteristics of the data, but often require an analyst to infer field semantics manually. Sequence alignment techniques including ones used in NETZOB [20] and NETPLIER [64], identify field boundaries by aligning common byte values across a collection of messages, but similarly burden analysts. Semantic approaches, such as those used by AWRE [53] and FIELDHUNTER [17], look for regions of messages which match a property, such as message length or hostname, to identify field boundaries and their types. While the descriptions from semantic methods can be used by the analyst directly, they do not generalize well to previously unseen data types. Security tools for reverse engineering protocols generally use only one of these approaches. BINARYINFERNO is instead built around a novel integration algorithm, which allows it to combine semantic inference, principled heuristics, and a semantically meaningful serialization search. This architecture benefits the analyst by providing semantically meaningful results, generalizability for unseen data types, and accurate inference of semantics for variable-length payloads.

Our approach to field inference assumes an error-free collection of messages, such as is provided by data transmitted with modern link and transportation protocols. We assume that fixed-width fields and variable-length payloads–composed of one or more fields–start at fixed offsets from the start of a message. This representation choice facilitates efficient and unambiguous deserialization, a desirable feature for exchanging binary data that is widely used, for example in IP [22], UDP [54], and BGP [55] protocols and ASN.1 BER serializations [37], [30]. To handle cases where this assumption does not hold, such as protocols with union types, BINARYINFERNO is tuned to not lead analysts astray by avoiding false positives. In Figure 1 we illustrate an example of a top-level protocol format with common header fields and two encapsulated payload formats.

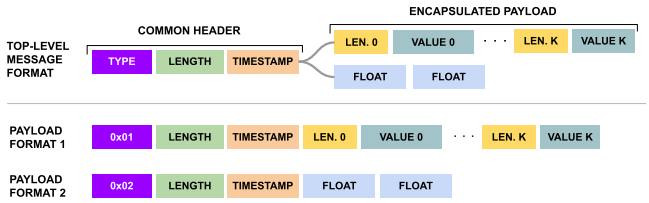

Fig. 1: Example of top-level message format with common header fields and two encapsulated payload formats.

BINARYINFERNO tackles the fields which are common across messages. In this manner, our approach infers those fields it can recognize and lets analysts focus on the remaining unrecognized regions; the analysts may have some insights into the data that help them partition unrecognized regions into groups before rerunning BINARYINFERNO to get more information. We hypothesize that developing an effective inference technique for fields common across formats will be a useful tool in its own right and will also enable us to bootstrap a recursive inference process to perform inference over formats with unions in future work. Finally, as in the related work, we assume fields are byte-aligned, with a minimum width of one byte, and that the data is not compressed or encrypted.

III. ILLUSTRATIVE EXAMPLE

Imagine you are an analyst reverse engineering a message format from a static network trace. A priori, you know the approximate times the messages were captured, but nothing else. Given the following three messages, which we use as a running example, your job is to recover the underlying format.

```
01000D60A67AED054150504C45
01000E60A67AF9064F52414E4745
01001160A67B0504504C554D0450454152
```

The unknown format has the following structure: first a 1-byte message type, then the 16-bit message length, next a 32-bit Unix timestamp, and finally a payload of one or more length-value pairs (Format 1 in Figure 1). BINARYINFERNO automatically infers exactly this description from these messages in about the time it takes to read this sentence. Figure 2 shows BINARYINFERNO's output on these messages.

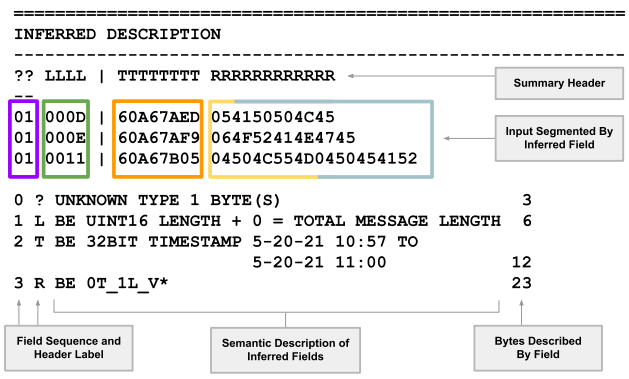

Fig. 2: BINARYINFERNO output for our running example.

IV. OVERALL APPROACH

Our key insight in tackling the field inference problem is that while there may be an infinite number of ways data can be encoded in binary, in practice engineers reuse standard encodings over and over again. That observation is true at both the level of atomic types, such as integers, IEEE 754 floats [14], and timestamps, and in the ways variable-length values are serialized. These common idioms leave behind fingerprints we can use to identify them. Typical idioms for representing variable-length sequences in binary include type-length-value (TLV) and length-value (LV) encodings, both of which include an explicit representation of the length of the sequence before the sequence itself appears. We can use this fingerprint to find a length field by correlating candidates with the length of the sequence.

As we consider a message sample looking for fingerprints, we collect two kinds of evidence. The first is within a single message: does the data at a particular offset look like an integer or float, could it plausibly encode a length field, *etc.* The second looks across messages to correlate the evidence collected in each message. If every message supports a given offset being a length, then we have strong evidence that the value at that offset is, in fact, a length.

The specific evidence that needs to be collected depends upon the hypothesized field type at a particular offset, and so we created an ensemble of detectors, each specialized to a particular kind of data. Each detector is responsible for analyzing the sample messages and returning a partial description in a data description language, identifying if and at what offsets its kind of data is likely to appear.

We have built a number of such detectors, including *atomic detectors* for identifying base fields such as IEEE 754 floats and timestamps; a *field-boundary detector* that uses changes in entropy to identify where one field ends and another begins; and a *pattern-based detector* that use iterative deepening search guided by common serialization patterns to infer variable-length fields. This collection of detectors is easily extended with new detectors.

Once the individual detectors have returned their findings, we automatically integrate the results into a single description that accurately describes as much of the sample as possible. The key challenge in integrating the results is resolving conflicts in which different detectors make conflicting claims about particular bytes in the message. We resolve such conflicts using an integration algorithm which maximizes the amount of sample data described as discussed in Section VIII. This approach means that the system is resilient to individual detectors making mistakes, and conversely, detectors can return partial descriptions corresponding even to weak signals, confident they will be overruled if another detector finds a stronger signal. We designed this integration strategy to maximize our ability to make progress on an unknown format by returning a partial description in the case where some fields in the format cannot be inferred because of limitations of the sample, or in situations where some field types are completely new to the tool. The resulting description can be used to parse the sample data and – to the extent that the inference was correct – any future data from the same source.

TABLE I: Binary arrangement of IEEE 754 32-bit Float. S labels the sign bit. E labels bits of the exponent. M labels bits of the significand.

Byte 0	Byte 1	Byte 2	Byte 3
S E E E E E E E	E M M M M M M M	M M M M M M M M	M M M M M M M M

V. Atomic Detectors

Each atomic detector is responsible for identifying a particular type of data. Each such detector takes the collection of sample messages as input and returns a partial description for any fields it recognizes. Internally, such a detector works by iteratively passing different byte slices to an inference function. We define a slice as a selection of bytes with a common offset and a common width from each message in a collection. The inference function returns a binary decision as to whether a field of the detector's type can be inferred at the supplied slice. By running a detector's inference function at each offset from the message start, we collect the set of all locations that may contain the data type in question. To provide a flavor for a range of atomic detectors, we describe the inference functions of three detectors in our ensemble.

A. Float Detector

Our float detector uses a heuristic-based approach that exploits distribution characteristics of the IEEE 754 floating point representation (floats) [14]. The key insight is that float values within a message set tend to be clustered in a relatively small range; we use this intuition along with observations on the structure of IEEE floats to detect likely candidates.

IEEE 754 Floats are composed of three regions: a single sign bit, an 8-bit binary encoded exponent, and a 23-bit binary encoded mantissa or significand. We illustrate the format in Table I. A real number value is represented approximately by a float in the following form: sign $\times$ 2^{exponent} $\times$ significand.

We observed that floats in real-world data sets [11] are often concentrated in small ranges relative to the entire range of values an IEEE 754 float can represent. When encoded as IEEE 754 floats, this concentration causes adjacent real values to use the same exponent value, but with distinct significand values. Through exploratory visualization, we observed that reuse of exponent values caused the frequency of 1 bits used in the 8 exponent bit positions to be greater than the average frequency of 1 bits used in the 23 bit positions of the significand for many of the real-valued dimensions encoded as binary floats. This frequency variation means that plotting a histogram of set bits in the exponent and significand forms a distinctive L-Shape, with the long part of the L lying down, as illustrated in Appendix A Figure 8. This shape is notably missing when plotted for other types of data, such as unsigned 32-bit integers.

We calculate a numeric measure of the L-Shape by taking the average frequency of the 1 bits in the significand and dividing it by the maximum frequency of 1 bits in the exponent. We call this measure the *L-Ratio*. The intuition behind measuring the maximum frequency for the exponent is that if a collection of floats all use the same non-zero exponent, then some bit position in the exponent will have a frequency of 1 bits equal to the number of floats in the collection. Further no bit position can have a frequency of 1 bits greater than the number of floats in the collection as each float can only contribute at most a single 1 bit in each position. The intuition behind measuring the average frequency of 1 bits in the significand is that significand values vary greatly compared to the corresponding exponents values. We use an average to smooth the significand 1 bit frequencies. Taken together, these two measures capture the balance between variety of significand values and reuse of exponent values over a collection of floats. We discuss selecting cut-off points for the L-Ratio and the datasets used in Appendix A.

As a 32-bit float is expressed across 4 bytes, with the significand entirely in the last 3 bytes, it is highly unusual to have a constant value across nibble-width slices in any of the last 3 bytes. We informally call such a feature a *constant stripe*. We reject as possible floats any slices that with constant stripes in the bytes of the significand.

For our running example, the float detector identifies bytes 4, 5, 6, and 7 across all three messages (`0xA67AED05`, `0xA67AF906`, `0xA67B0504`) as a potential float field; they are all three valid, normalized floating point values clustered around $-8.7x10^{-}16$. The integration algorithm will ultimately overrule the float detector, as other explanations allow more data in total to be described across the entire set of messages.

B. Timestamp Detector

Our approach to timestamp inference assumes knowledge of when a static network trace was captured. Tools like Wireshark [12] and tcpdump [9] capture this information automatically. Alternatively, an analyst's rough idea of when a sample was captured can be utilized.

Given a start and end date-time for the capture, the timestamp detector interprets the slice values according to one of the timestamp formats it supports. If all slice values interpreted according to the specific timestamp format are within the specified range, we infer a timestamp field[1]. If any values fall outside of the range, we move on. We summarize this detector in Equation 1 in which $\mathbf{X}$ is the slice under consideration, and the function f interprets slice value x according to a specific timestamp format.

$$isTimestamp(\mathbf{X}) = \bigvee_{x \in \mathbf{X}} \text{start} \leq f(x) \leq \text{end} \tag{1}$$

Our timestamp detector implementation is modular, requiring only a format epoch value and a function to extract the format offset in seconds. Our abstraction makes it easy to add detectors for new timestamp formats. By default, our timestamp detector has inference functions for NTP and Unix epoch timestamp formats. For NTP timestamp formats, our detector differentiates between NTP timestamps with null values and pure NTP timestamps where every value is within the range.

For our running example, the analyst knows the approximate time the network trace was captured and provides it to BinaryInferno which parameterizes the timestamp detector. The detector's Unix epoch inference function identifies the

[1]To account for systems with an unknown timezone offset, the timestamp detector automatically expands the start and end date-time by 24 hours.

slice comprised of bytes 3, 4, 5, and 6 as a valid Unix epoch timestamp with all values within the provided span and infers a Unix timestamp field for this slice. This inference is correct; we discuss why this result "wins" during the merge with other possible descriptions of these bytes in Section VIII.

C. Length Detector

A common idiom in binary packet data is to encode variable-length data by first encoding the unsigned integer length of the data followed by the data itself. Our length detector searches for slices which exactly explain the lengths of a collection of messages. We call these *strict length fields*. We consider a slice $\mathbf{X}$ to exactly explain the lengths if the unsigned integer interpretation of each slice value x_i plus some non-negative constant k equals the corresponding the message length ($\mathbf{L}_i$). The intuition is that the values in $\mathbf{X}$ correspond with message lengths, while k describes a fixed header size. We summarize this boolean function in Equation 2.

$$isStrictLength(\mathbf{X}, \mathbf{L}) = \bigvee_{x_i \in \mathbf{X}} ((x_i + k) = \mathbf{L}_i) \tag{2}$$

We can calculate this value by taking the intersection over the differences between the slice values and message lengths: $\bigcap_{x_i \in \mathbf{X}} (x_i - \mathbf{L}_i)$. If the resulting set has only a single non-negative value k, then we infer a length field with the value k as our constant. We discuss the probability of random data being mistaken for a length field in Appendix B and *fuzzy lengths* –fields which do not directly encode the length of the message– in Appendix C. For our running example (Figure 2), our algorithm infers two possible length fields. bytes 1 and 2 taken together (`0x000D,0x000E,0x0011`) and byte 2 in isolation are consistent with the overall message lengths of 13, 14, and 17.

VI. Field-Boundary Detector

We derive field boundaries in three ways. First, through the direct recognition of the field, either by an atomic detector (Section V) or by a pattern-based detector (Section VII). Second, through the recognition of adjacent fields using the same mechanisms. Finally, by using a stand-alone *field-boundary detector*, which identifies boundaries without recognizing fields directly using an information theoretic approach.

The inspiration for the field-boundary detector was the observation that multi-byte fields often contain different amounts of information, as defined by Shannon entropy [57], in the various bytes of a slice. For example, we have observed that the least significant byte of an integer field will often have more information than the most significant byte: the least significant bits vary frequently, while the most significant bits vary only when values differ by an order of magnitude or more. Similarly, the least significant byte of an IEEE Float containing a portion of the significand varies more than the most significant byte of a IEEE Float containing the exponent.

Leveraging this intuition, our field-boundary detector looks for adjacent 1-byte slices that exhibit a significant difference in their Shannon entropy, calculated using the formula

$$H(\mathbf{M}_k) = -\Sigma_{v \in \mathbf{M}_k} P(v) \, log \, P(v) \tag{3}$$

where $\mathbf{M}_k$ is the bag formed by the kth byte of each message in the sample $\mathbf{M}$, and $P(v)$ is the probability of value v in $\mathbf{M}_k$ calculated as the number of times v appears in $\mathbf{M}_k$ divided by the size of $\mathbf{M}_k$. We choose 1-byte slices as larger slices risk mistakenly interpreted the entropy of multiple adjacent fields is as belonging to a single field.

As with our other detectors, the field-boundary detector assumes a fixed endianness for the entire collection of messages. Unlike most other detectors, it takes the endianness into consideration when deciding whether a boundary is likely to occur between adjacent slices. When considering data assumed big-endian, for any pair of adjacent 1-byte slices $\mathbf{A}$ and $\mathbf{B}$, we infer a field boundary if $H(\mathbf{A}) - H(\mathbf{B}) \geq 1.0$, reflecting the intuition we expect more information in the earlier slice under a big-endian encoding. For data assumed little-endian, the situation is reversed, and so we infer a boundary if $H(\mathbf{B}) - H(\mathbf{A}) \geq 1.0$. In both cases, we chose a threshold of 1.0 as this value requires the more informative slice to have twice as much information regardless of the number of messages.

In our running example, the field-boundary detector infers a field boundary between bytes 2 and 3 as shown in Figure 2. For byte 2, the Shannon entropy, calculated as H([`0x0D,0x0E,0x11`]), is 1.584, and for byte 3, it is H([`0x60,0x60,0x60`]), which is 0.0. Their difference is 1.584, well above the threshold.

VII. Pattern-based Detector

Not all data fits into fixed-width fields: strings, lists, *etc.* With fixed-width fields, only an offset is necessary to decide how to interpret a value. In contrast, data in variable-length fields needs to be encoded in a way that allows the reader to understand where the field ends; we call these encoding conventions *patterns*. For example, one common serialization pattern is the length-value (LV) pattern, which describes a data value by prepending it with information about its size. Another common pattern is the type-length-value (TLV) pattern, which prepends the length-value pattern with information about the protocol-specific semantic type of the value.

We can combine a detector based on these two patterns – LV and TLV – with our other detectors to infer comprehensive message format descriptions on the basis of limited sample data. These patterns also allow us to recover "hidden" fixed-width fields by treating the variable component of a message as a discrete section and re-aligning our sample appropriately. The resultant portions are then amenable to inference using our atomic and field-boundary detectors.

To build a detector for variable-length data, we describe these patterns in a grammar and then perform an iterative deepening, depth-first search [40] over strings in our grammar, searching for patterns that explain the sample messages. Iterative deepening operates by performing a bounded depth-first search over strings in the grammar up to a certain size. If no consistent format string is found within that depth, the search is rerun to an increased depth. As a result, we discover simpler format strings sooner, and we can guarantee we will explore all formats up to a specified complexity.

Representing Patterns. We represent message formats in EBNF as shown in Grammar 1. The grammar contains a single

terminal BYTE that denotes an 8-bit binary string. We enclose non-terminals in angle brackets. We use integer superscripts to indicate a number of repetitions, with parentheses denoting the scope of the superscript as necessary. If the superscript is an earlier field, we surround the superscript with square brackets to indicate the number of repetitions is dependent upon the value of the earlier field. When square brackets are omitted in the superscript, the number of repetitions is assumed to be constant for all messages.

Each string in the grammar characterizes a specific message format employing a common binary serialization pattern; each token in such a format string represents a specific field. A collection of messages is consistent with a format string if the format accounts for every byte in every message, precisely explaining all variation in message lengths.

$$
\begin{array}{lll}
\langle PATTERN \rangle & ::= & L(V)^{[L]} \\
 & | & TL(V)^{[L]} \\
\langle T,L,Q \rangle & ::= & \texttt{BYTE}^{N}\ \texttt{(1 } \leq \texttt{ N } \leq \texttt{ 4)} \\
\langle V \rangle & ::= & \texttt{BYTE} \\
\langle FIELD \rangle & ::= & \langle VLFIELD \rangle \\
 & | & \texttt{BYTE}^{P}\ \texttt{(P } \geq \texttt{ 1)} \\
\langle VLFIELD \rangle & ::= & \langle PATTERN \rangle \\
 & | & Q(\langle PATTERN \rangle)^{[Q]} \\
 & | & Q(VV)^{[Q]} \mid \ldots \mid Q(VVVVVVVV)^{[Q]} \\
\langle FORMAT \rangle & ::= & \langle FIELD \rangle \\
 & | & \langle FIELD \rangle\ \langle FORMAT \rangle
\end{array}
$$

Grammar 1: Serialization Pattern Description Grammar

The key to the grammar is the ⟨*PATTERN*⟩ non-terminal that represents the two serialization patterns described earlier: length-value (LV) and type-length-value (TLV). Non-terminals ⟨*T*⟩, ⟨*L*⟩ and ⟨*Q*⟩ describe the type, length and quantity aspects of serialization patterns. We restrict these meta-variables to be BYTE fields from 1 to 4 bytes long as is consistent with real-world use. We use meta-variable ⟨*V*⟩ to describe the payload.

As an example, we write the 1-byte length-value pattern as $L(V)^{[L]}$. This pattern interprets the byte-string 0x03AABBCC by assigning *L* the value 3 and the grammar component $(V)^3$ the sequence of bytes 0xAABBCC. In contrast, byte strings 0x03AABB and 0x03AABBCCFF are inconsistent with the pattern $L(V)^{[L]}$. The first fails to provide enough data while the second provides too much data.

Non-terminal ⟨*FIELD*⟩ indicates a field can be varying in length (⟨*VLFIELD*⟩) or fixed-width ($\texttt{BYTE}^{P}$). Non-terminal ⟨*VLFIELD*⟩ can be a pattern, a repetition of a pattern with a quantity given by preceding value ⟨*Q*⟩, or a repetition of 2 (*VV*) to 8 byte (*VVVVVVVV*) words whose quantity is given by the preceding value ⟨*Q*⟩. Non-terminal ⟨*FORMAT*⟩ concatenates fields, allowing us to describe message formats where the variation in message length comes from more than one pattern, as well as formats with fixed-width prefixes and suffixes.

Pattern Inference. Our depth-first search proceeds in the following manner. Given a set of messages, we attempt to interpret each message according to a format string *F* drawn from the ⟨*FORMAT*⟩ branch of the grammar. We first deserialize the message according to *F* from the starting offset and produce a new offset *end*. We consider a message *interpreted* if *F* does not attempt to read beyond the available bytes in the message, that is to say if the resulting offset *end* is less than the length of the message in bytes. We consider a collection of messages interpreted by *F* if each message is interpreted. In the case that *end* is strictly less than the message length for some subset of messages in the collection, we infer that we are in the second branch of ⟨*FORMAT*⟩ (*i.e.*, ⟨*FIELD*⟩ ::= ⟨*FIELD*⟩ ⟨*FORMAT*⟩) and our detection recurs on the remaining bytes in the message. If the collection of messages is not interpretable, because a pattern requires more bytes than some message has, we try the next format string drawn from ⟨*FORMAT*⟩.

When we have exactly interpreted all bytes across the collection of messages by locating a format consistent with the data, we report that format as an inferred description for the entire collection. If we have exhausted all format strings in the grammar up to the byte-length of the shortest sample message, but we have not found an acceptable interpretation, we report no consistent pattern could be found. The search algorithm continues until we find a configurable minimum number of descriptions, or we have considered all format strings.

Implementation Optimizations. Our search uses three optimizations. First, while performing the search, we want to avoid repeatedly interpreting the same message according to the same pattern at the same offset. To prevent this, we memoize the possible interpretations of ⟨*VLFIELD*⟩ at every offset from message start prior to performing the search. For each pattern, message, and starting offset, we store the ending offset generated by interpreting the message or a nil value if the pattern attempts to read beyond the available bytes. We refer to this pre-evaluated set as **MEMOS**[*pattern*, *message*, *offset*].

Second, constant-valued slices in a collection of messages occur with some regularity. To avoid spuriously inferring patterns where none exist, we use a heuristic to restrict their influence. Consider an example where a 1-byte slice has the value 0 for every message. While this slice is consistent with a length-value pattern – it represents a length value of 0, followed by no data – we choose to ignore it as it offers no evidence that would support the inference of this pattern. As another example, consider an example where a slice has the value 4 for every message. If this slice were part of a length-value field, the values would all have the same length (4) and would be better explained as a fixed-width 5-byte field. Accordingly, if the portion of a format string that controls the number of repetitions (⟨*L*⟩ or ⟨*Q*⟩ in the grammar) is a constant, we skip interpretation and instead allow the search algorithm to consider the next format string in the grammar.

Third, the problem is embarrassingly parallel. By partitioning how we draw format strings from the grammar we can conduct the search in parallel over multiple processors.

Star Patterns. A common variation on serialization patterns is for the quantity of pattern repetitions (*Q* in $Q(\langle PATTERN \rangle)^{[Q]}$) to be omitted. This pattern form is present in bgp open messages and dhcp options, among others. We call such cases *star patterns* based on their relationship to the Kleene star operator. We define star patterns, ⟨*STAR-PAT*⟩ in

Grammar 2, as a ⟨*PATTERN*⟩ repeated zero or more times with three restrictions. First, the pattern must occur at least once in one of the sample messages. This restriction prevents the consideration of zero repetitions of patterns across a collection of messages, that is to say, inferring a star pattern at every offset. Second, the number of repetitions of a pattern must vary in at least one of the messages. This restriction prevents the star pattern from being inferred when the data could more simply be explained by a fixed sequence of patterns. For example, without this restriction a collection of messages consistent with format $L(V)^{[L]}L(V)^{[L]}$ would also be consistent with $L(V)^{[L]}{}_*$. We choose to infer a star pattern only when the data cannot be described otherwise. Finally, we infer star patterns only when they are the final variable-length portion of a message format. This requirement is consistent with the common usage of star patterns in the real-world message formats we have examined. We allow for a constant fixed-width field following the star pattern, such as a checksum, with the term $\texttt{BYTE}^S$.

$$\langle \textit{STAR-PAT} \rangle ::= \langle \textit{PATTERN} \rangle^* \ \texttt{BYTE}^S \ \texttt{(S} \geq \texttt{0)}$$

$$\begin{aligned} \langle \textit{STAR-FMT} \rangle ::= & \ \langle \textit{FORMAT} \rangle \\ | & \ \langle \textit{FORMAT} \rangle \ \langle \textit{STAR-PAT} \rangle \\ | & \ \langle \textit{STAR-PAT} \rangle \end{aligned}$$

Grammar 2: Star Pattern Description Grammar

Inferring star patterns. Intuitively we define a star pattern as being inferred for a collection of messages if the repeated interpretation of the pattern ⟨*PATTERN*⟩ across each message can derive a single offset relative to message end across the entire collection. More precisely, starting from some offset, we want to calculate all the offsets we can reach by repeatedly using the pattern. If we explain all remaining bytes in each message, or all but a constant number of bytes (S), we infer a star pattern. We discuss the details of identifying star patterns in Appendix D.

Our pattern search method proceeds unchanged, drawing format strings starting with ⟨*STAR-FMT*⟩ in Grammar 2 and performing lookups in **MEMOS**. While not shown here, this approach can be easily extended to allow some messages in the collection to have zero instances of a pattern.

VIII. Integration Algorithm

At this point, our ensemble of detectors has inferred independent partial descriptions of the sample under study, where a partial description is any set of inferred fields and stand-alone field boundaries. The detectors in our ensemble are independent and can be run in any order as they exchange no information. The next step is to combine these partial descriptions into a single description that accurately describes as much of the sample as possible, resolving any conflicts. Conflicts arise when two or more inferred fields make differing claims about the same bytes; for example, when one inferred field claims that bytes 0 through 3 are an integer, while another claims that bytes 2 through 5 are a float. These fields conflict because they make incompatible claims about bytes 2 and 3. To handle these cases, we devised a graph-based algorithm to heuristically find the highest confidence, most comprehensive, conflict-free description that is compatible with the inferred set of partial descriptions. Specifically, we construct a directed acyclic graph (DAG) whose nodes (**V**) are the inferred fields (plus special Source and Sink nodes) and whose edges (**E**) capture a "strictly precedes" relationship between the corresponding fields. The special Source node strictly precedes all others, while all other nodes strictly precede the special Sink node. We treat a partial description inferred from a pattern-based detector as a single field.

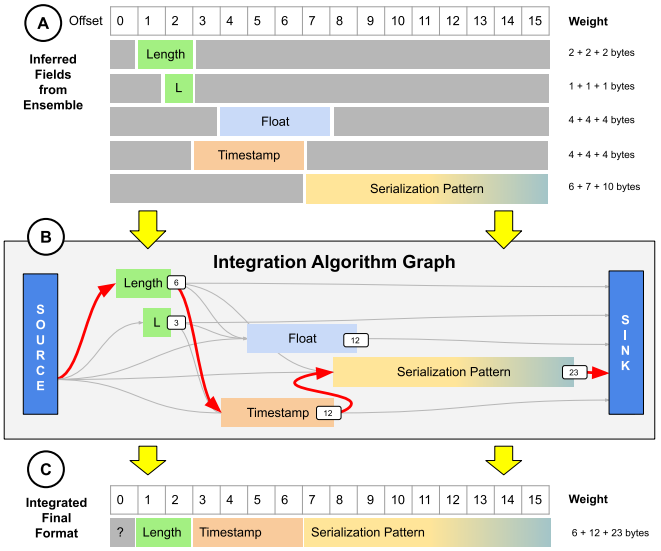

Fig. 3: Conflict resolution process for our running example. Each inferred field in (A) becomes a node in the Integration Algorithm Graph (B) where topological sort is used to introduce edges between conflict free pairs of nodes. The optimal description (C) is found by calculating the maximum path (shown in red) from source to sink.

In Figure 3 we show an example of constructing a DAG from a collection of all partial descriptions inferred from the three messages in our running example. The inferred float field has a weight of 12 bytes (4 bytes in each of three messages). The 2-byte length field by contrast has a weight of 6 bytes. We would assign any stand-alone field boundaries, which have a zero-byte width, a weight of 1.0. We assign the Source node a weight of 0. (Edges take their weights from their sources, so the weight of the Sink is irrelevant.) In Figure 3, conflicts between the two length fields, between the timestamp and float field, and between the float field and the serialization pattern are shown. These conflicts are resolved to maximize the weight (number of bytes) described overall.

Given this encoding, each path through the DAG from Source to Sink represents a conflict-free sequence of inferred fields. To give preference to paths that correspond to better descriptions, we assign a weight to each edge, computed as how many bytes in each message that the sample field e_s covers in each of the messages. In other words, we calculate the weight of a field as the total number of bytes it describes across all messages in the sample. With these weights, finding the best conflict-free description reduces to returning the path through the DAG with the maximum weight. This search can be done in $O(|\mathbf{V}| + |\mathbf{E}|)$ time by first topologically sorting the

nodes by the strictly precedes order. In the rest of this section, we describe this algorithm in more detail.

Encoding conflicts. We use the strictly precedes relation to judge whether two inferred fields are conflict-free. For each field inferred from a collection of messages, there is a corresponding interval in each message, spanning from the start to the end offsets for that field in that message. We say that field **A** *strictly precedes* field **B**, written $\mathbf{A} < \mathbf{B}$, if each **A** interval ends before the corresponding **B** interval begins. Effectively we require that in every message in the collection, all bytes in field **A** end before all bytes in field **B** begin.

More precisely, we use interval ordering semantics with a closed start interval and an open end interval. We consider an interval $[a_{start}, a_{end})$ well-formed if $a_{start} \leq a_{end}$. For all pairs a and b of well-formed intervals from fields **A** and **B**, respectively, $a < b$ iff $[a_{start}, a_{end}) < [b_{start}, b_{end})$, which is to say that if for all such pairs, $a_{end} \leq b_{start}$, then $\mathbf{A} < \mathbf{B}$. Intuitively, fields which are strictly disjoint are orderable in the strictly precedes relation, while fields which share bytes are not. (By our assumption that all messages in our sample have a consistent format, field order will be consistent across all messages for any pair of fields in the strictly precedes relation.)

Our notation for field intervals includes stand-alone field boundaries as zero-width intervals. Consider the example field **A** such that each interval is $[a_{start}, a_{end})$, and stand-alone field boundary **B** $[b_{start}, b_{end})$ where $b_{start} = b_{end} = a_{end}$. By our definition of strictly precedes, $\mathbf{A} < \mathbf{B}$. This flexibility allows our graph algorithm to consider both full fields as well as stand-alone field boundaries. In this way, stand-alone field boundaries that fall between two full fields act as a reinforcement when considered by our algorithm.

Calculating the maximum path. Finally, we find the heuristically best description by computing the maximum path through the weighted DAG by recursively relaxing the graph, node by node, in topological order sorted by the strictly precedes relation starting at the Source node. Intuitively, the maximum path is the sequence of fields related by the strictly precedes relation that maximally describes the information in the sample when evaluated by our ensemble. Our definition of strictly precedes and our construction of the DAG ensures that all the fields are disjoint. To extract the fields on the maximum path, we start at Sink and repeatedly follow the maximum inward pointing edge until we arrive at Source. Reversing that list gives the final inferred description. Alternatively, if we desired multiple descriptions, we could use Yen's algorithm [65].

IX. Evaluation

We evaluate BinaryInferno's ability to provide useful information to an analyst and compare its performance to that of related state-of-the-art tools. Recall that the goal is a system that saves the analyst time by producing precise, semantically meaningful descriptions with low false positive rates for inferred field boundaries.

What we compare against. Our security use case is that of an analyst attempting to reverse engineer an unknown protocol to identify vulnerabilities such as in a proprietary system. We compare BinaryInferno to five state-of-the-art protocol reverse engineering tools capable of field inference: Awre [53], FieldHunter [17], Nemesys [38], Netplier[64], and Netzob [20]. Awre identifies fields according to semantic types drawn from wireless radio message formats and produces a semantic description. FieldHunter similarly identifies fields using semantic types common to network protocols. Nemesys uses a heuristic approach to identify field boundaries based on bit-level differences between adjacent bytes, while Netplier employs a string alignment heuristic. Netzob is a Python based interactive environment designed to support an analyst performing manual reverse engineering. For our evaluation we use Netzob's implementation of the Needleman & Wunsch sequence alignment technique [51] to identify field boundaries. We use a public re-implementation of FieldHunter[2]. We compare how well these tools identify field boundaries in unknown protocols as this is an essential step [39] in reverse engineering the format.

Benchmark suite. Our sample data was drawn from a broad collection of data representing different types of top-level binary protocols including those explored in related work. We selected well-known network protocols from networking, industrial control, UAV systems, and malware as these are all areas in which protocol reverse engineering is performed for security purposes. Our goal is to characterize BinaryInferno's performance on binary protocols generally, rather than on a specific type. Thus, our benchmark suite is comprised of five common binary protocols (bgp [55], dhcp [29], ntp [46], smb [47], and smb2 [48]), two industrial control protocols (dnp3 [13] and modbus [49]), two specialized binary protocols (mavlink [16] , the mirai botnet's command & control traffic [15]), and one protocol created as part of a reverse engineering tutorial [7].

For each top-level protocol in our evaluation, we collected one or more representative sample traces to build a collection with at least 1000 messages. For protocols bgp, dhcp, ntp, smb, and smb2, we use traces collected from network security competitions [8], public reference traces [1], [4], [6], [10] and the NetPlier test suite [5]. For dnp3 and modbus we use captures from security researchers and network security competitions [1], [36]. For mavlink, we use a trace from a software-in-the-loop (SITL) drone simulator [3]. For mirai we use a trace captured from a virtualized instance of a mirai botnet used for security research [23].

For the tutorial example, we directly transcribed sample messages from a network protocol reverse engineering tutorial website [7] and generated additional messages by encoding common dictionary words in accordance with the specification. This sample is notable for representing a baseline for evaluating different protocol reverse engineering tools against a human expert. Specifically, it is simple enough by construction that a human expert can readily reverse engineer the format. Tools which have difficulty with this "Hello World" format may be of limited use when presented with more complicated real-world formats.

For each protocol trace, we extracted all packets of the specific protocol, and then randomly drew 1000 messages (without replacement) to create our initial sets of 1000 messages per protocol. We used the first 500 and 100 messages for our smaller subsets.

[2]https://github.com/vs-uulm/fieldhunter

To create additional test sets for evaluating field inference in payloads, we selected the two top-level protocols containing union types over encapsulated payload formats (bgp, mavlink), and binned the messages into subsets by payload format. These payload samples permit us to evaluate the feasibility of using BinaryInferno to differentiate payload formats once a message type field is identified. To create these subsets, we binned the original sample messages by the message type field according to the ground-truth formal specification and randomly drew up to 1000 messages from each bin.

Establishing ground-truth. For each protocol we determined the ground-truth field boundaries from reference literature. We then created a formal specification, which we used to parse and segment message bytes into discrete fields. Consistent with Netplier's evaluation, when adjacent fields in a sample had constant values across all messages, we updated our ground-truth to merge these fields.

Processing and extraction. As input BinaryInferno takes an ASCII file with the hexadecimal value of each binary message on a separate line. We chose this input format to facilitate ease of use and portability independent of the message source. To extract messages from static network traces (`pcaps`) into our input format, we use the Wireshark's [12] `tshark` utility. We constructed harnesses for Awre, FieldHunter, Nemesys, Netplier and Netzob to use the same input format and to return messages segmented into fields as output.

The `dhcp` message format allows the end of messages to be padded with an arbitrary quantity of zero-valued bytes. The use and quantity of this padding is unspecified. We use an automatic method to detect padding and remove it prior to inference as described in Appendix E.

BinaryInferno can leverage two pieces of analyst-supplied prior knowledge to refine its inference: the assumed endianess of the sample and the approximate date range when the sample was collected. We include the results of inference using the endianness and the date range of the sample capture as **BI+** in our evaluation. We label the result without the use of these priors as **BI**. While the standard [56] for data transmitted over the internet is big-endian (network byte order), there is no strict requirement that protocols adhere to it.

Execution environment. We ran all experiments on a Linux server equipped with 40 CPU cores (Intel® Xeon® Gold 6140 CPU @ 2.30GHz) and 128GB memory. We allowed each tool up to 60 minutes of execution time per sample. We make all samples used in our evaluation available to the public [2].

A. Evaluating Results

For each sample in our evaluation we calculated the precision, recall, false positive rate, and F1 score for each tool's result relative to the ground-truth format. These measures represent standard evaluation metrics for tools like BinaryInferno, and they match the metrics reported by similar systems.

We define *positives* to be field boundaries in the ground-truth format. We define inferred true field boundaries, or true positives (TP), to be those boundaries inferred by a tool which match the ground-truth specification, as illustrated in Figure 4. We define *negatives* to be adjacent bytes belonging to the same field according to ground-truth, such as between byte offsets 0 and 1 in the Figure.

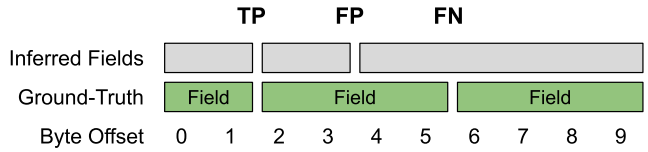

Fig. 4: Illustration of true positives (TP), false positives (FP) and false negatives (FN) between inferred field boundaries and ground-truth.

$$Precision = \frac{\text{Inferred True Field Boundaries (TP)}}{\text{Inferred Field Boundaries (TP + FP)}} \quad (4)$$

$$Recall = \frac{\text{Inferred True Field Boundaries (TP)}}{\text{True Field Boundaries (positives)}} \quad (5)$$

$$FPR = \frac{\text{Inferred False Field Boundaries (FP)}}{\text{Adjacent Field Bytes (negatives)}} \quad (6)$$

We calculate *precision* as the number of true field boundaries (TP) inferred, divided by the total number of field boundaries inferred (TP + FP) as shown in Equation 4. Intuitively, precision measures how likely something inferred as a field boundary is actually a field boundary. A precision of 1.0 (best) means that every inferred field boundary is a true field boundary, while a precision of 0 means no true field boundaries were inferred. We calculate *recall* as the number of true field boundaries inferred (TP), divided by the number of true field boundaries overall (positives) as shown in Equation 5. Intuitively recall measures the proportion of true field boundaries discovered. A recall of 1.0 (best) means the tool infers all true field boundaries while a recall of 0 means the tool infers none. We calculate false positive rate (FPR) as the number of incorrectly inferred field boundaries (FP) divided by the number of contiguous field bytes (negatives) as shown in Equation 6. A FPR of 1.0 means every inferred field boundary is incorrect, while a FPR of 0 (best) means every inferred field boundary is correct. For formats consisting of a single field, or formats where all fields are constant valued for a sample, we report precision of 1.0, recall of 1.0, and a FPR of 0.0 if no field boundaries are inferred by a tool. In the unusual case when a tool reports no fields inferred for formats with ground-truth field boundaries–returning the input unchanged–the precision of the result is undefined. We exclude such results from our averages in the interest of treating each tool fairly. We use the standard calculation of F1 score as the harmonic mean between precision and recall. F1 score balances precision and recall into a single metric for comparing overall accuracy between tools.

B. Testing on top-level protocol samples.

To evaluate BinaryInferno's ability to produce useful descriptions with low false positive rates, we ran the tool on our top-level protocol samples. For each sample we performed inference with the previously described data sets of size 1000, 500, and 100 messages to characterize the impact of sample

size on our results. Summary results of our top-level protocol evaluation for 1000, 500, and 100 messages are shown in Table II and plotted for 1000 messages in Figure 6. We report detailed results by sample size in Table V.

BINARYINFERNO significantly outperformed AWRE, NEMESYS, NETPLIER, and NETZOB on average precision and F1 score across all sample sizes. For example, BINARYINFERNO's average average-precision across the three sample sizes was 0.66 compared to 0.15 for AWRE, 0.31 for NEMESYS, 0.27 for NETPLIER, and 0.56 for NETZOB. BINARYINFERNO's average average-F1 score was 0.66 compared to 0.04 for AWRE, 0.34 for NEMESYS, 0.37 for NETPLIER, and 0.47 for NETZOB. BINARYINFERNO with priors (BI+) did even better. BI+ had the highest (best) precision rates (0.82 to 0.93) and the lowest (best) false positive rates (0.005 to 0.02), but even without priors, BINARYINFERNO's false positive rates (0.04 to 0.05) outperformed both NEMESYS (0.10 to 0.11) and NETPLIER (0.22 to 0.26) and tied AWRE.

NETPLIER exhibited the highest average recall (0.73 to 0.83) because of its tendency to divide messages into 1-byte fields, unsurprisingly finding the majority of field boundaries with that approach (but leading to a high false positive rate). BINARYINFERNO's performance improved on larger data sets, reaching 0.73 on the 1000 sample size.

FIELDHUNTER did not identify field boundaries in 69% of the samples given (40% of the top-level samples, 80% of the payload samples). FIELDHUNTER uses semantic types to perform identification, and accordingly has a low false positive rate for those types, but at the expense of generality. For top-level protocol samples where FIELDHUNTER made an attempt, it had precision of 0.68, recall of .37, and a FPR of 0.01. As field inference is an essential part of protocol reverse engineering, the utility of FIELDHUNTER to assist an analyst would appear limited as they would still have to reverse engineer the majority of the samples by hand.

To illustrate how these results compare from an analyst's perspective, we visualize the inferred field boundaries produced by each tool for the `tutorial` sample in Figure 5.

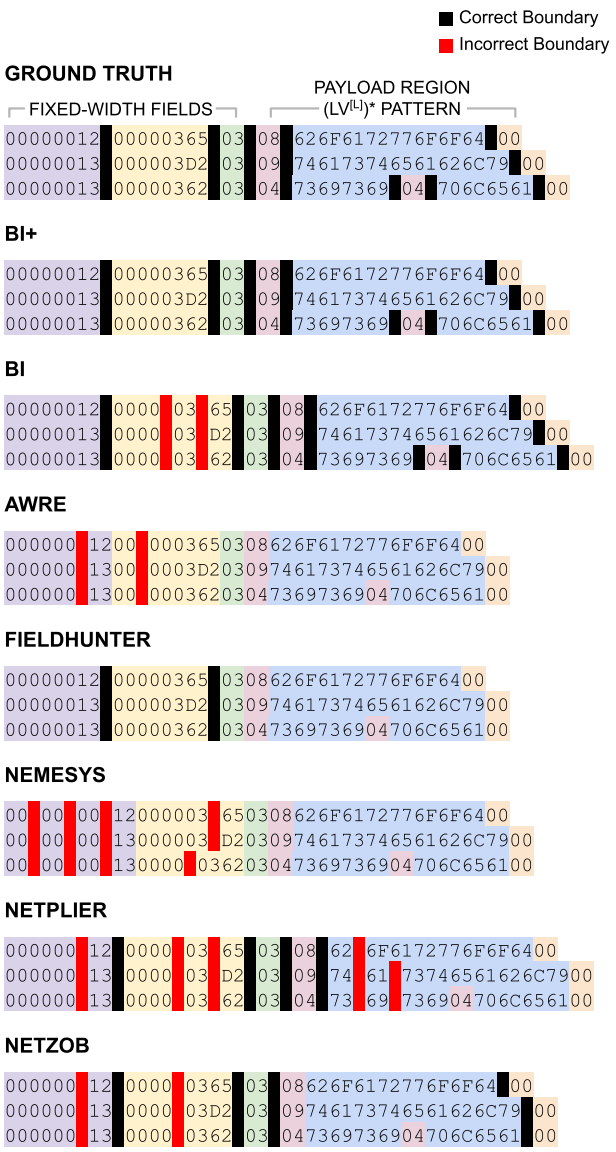

Fig. 5: Illustrated comparison of correct and incorrect inferred field boundaries using 1000 messages from the `tutorial` sample. Field data is color coded, with solid bars indicating correct (black) and incorrect (red) inferred field boundaries.

C. Testing on encapsulated payloads.

We hypothesize that BINARYINFERNO can be used to differentiate formats with unions by performing inference on each branch. To test this hypothesis, we evaluated BINARYINFERNO's performance on encapsulated payload types for two top-level protocols. For this evaluation we use previously described samples drawn from bgp and mavlink messages binned by message type . We summarize our encapsulated payload results in Table III, with detailed results in Table V.

Tools using semantic methods including AWRE, BINARYINFERNO, and FIELDHUNTER may not always infer a field format either due to limited data, or the lack of a semantic type in their ensemble, claiming each message is a single field. This behavior is one reason why these methods have markedly low false positive rates. For our 26 encapsulated payload samples, we excluded the empty results returned by these tools (1, 3, and 21 instances respectively) when calculating average performance.

For encapsulated payload samples, BINARYINFERNO (BI) had an average precision of 0.54, average recall of 0.65, an average False Positive Rate (FPR) of 0.08, and an average F1 score of 0.55. However, when supplied the underlying endianess as a prior and the approximate time of packet capture, BINARYINFERNO (BI+) had an average precision of 0.94, an average recall of 0.80, an average FPR of 0.01 and an average F1 score of 0.84. BINARYINFERNO (with and without priors) outperformed AWRE, NEMESYS, NETPLIER, and NETZOB on all metrics with the lone exception of NETPLIER's average recall of 0.78, which results from NETPLIER's tendency to divide messages into 1-byte fields. FIELDHUNTER does almost as well as BI+ but on a much smaller set of inputs: 5 out of 26 (versus 23 out of 26 for BI+). FIELDHUNTER had an average precision of 0.85, average recall of 0.78, an average FPR of 0.01, and an average F1 score of 0.80.

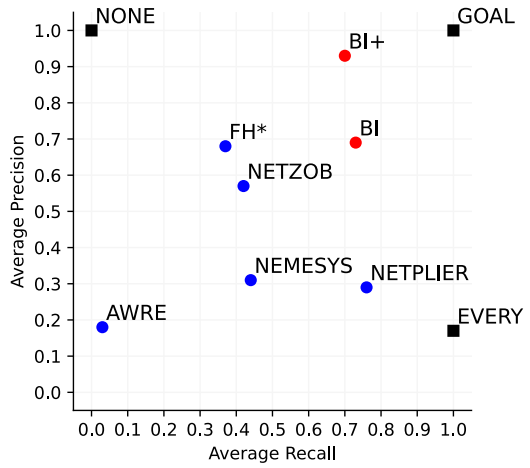

Fig. 6: Plot of average precision vs. recall on top-level samples of 1000 messages for AWRE, FIELDHUNTER, NEMESYS, NETPLIER, NETZOB, BINARYINFERNO (BI), and BINARYINFERNO with priors (BI+). Tools closer to ground-truth (GOAL) are better. Segmenting at each message byte is shown as EVERY. Segmenting nowhere in the message is shown as NONE. FIELDHUNTER (FH*) is plotted using only the 6 samples for which it identified fields.

TABLE II: Average precision, recall, FPR and F1 scores on top-level protocol samples by tool. Bold indicates best result.

Tool	# Msgs.	Prec.	Rec.	FPR	F1
BI+	1000	**0.93**	0.70	**0.005**	**0.78**
	500	**0.91**	0.69	**0.005**	**0.77**
	100	**0.82**	0.63	0.02	**0.70**
BI	1000	0.69	0.73	0.04	0.70
	500	0.69	0.72	0.04	0.69
	100	0.60	0.62	0.05	0.59
AWRE	1000	0.18	0.03	0.04	0.05
	500	0.18	0.03	0.04	0.05
	100	0.08	0.01	0.05	0.02
FIELDHUNTER*	1000	0.68	0.37	0.01	0.45
	500	0.68	0.37	0.01	0.45
	100	0.48	0.29	**0.01**	0.34
NEMESYS	1000	0.31	0.44	0.11	0.34
	500	0.31	0.45	0.11	0.34
	100	0.30	0.44	0.10	0.33
NETPLIER	1000	0.29	**0.75**	0.22	0.38
	500	0.27	**0.73**	0.22	0.37
	100	0.25	**0.83**	0.26	0.37
NETZOB	1000	0.57	0.42	0.03	0.47
	500	0.57	0.42	0.03	0.48
	100	0.53	0.45	0.04	0.45

* Results where FIELDHUNTER found no fields are excluded from averages.

D. Testing on Serialization Patterns

Compared to existing tools, BINARYINFERNO's pattern-based detector (Section VII) is uniquely able to infer exact serialization specifications from collections of messages by searching over combinations of common serialization patterns. NETZOB requires an analyst to manually infer and annotate segmented fields with semantics. NEMESYS and NETPLIER only produce segmented messages with no semantic description, while AWRE and FIELDHUNTER do not attempt this type of inference at all.

TABLE III: Average performance on payload samples. Bold indicates best result.

Tool	Precision	Recall	FPR	F1
BI+	**0.94**	**0.80**	**0.01**	**0.84**
BI	0.54	0.65	0.08	0.55
AWRE	0.14	0.11	0.11	0.11
FIELDHUNTER	0.85	0.78	**0.01**	0.80
NEMESYS	0.24	0.52	0.17	0.31
NETPLIER	0.30	0.78	0.24	0.40
NETZOB	0.47	0.53	0.07	0.47

TABLE IV: BINARYINFERNO Pattern-based detector results for top-level sample sizes of 100, 500, and 1000 messages. Y means that the ground-truth serialization pattern was inferred as the most likely pattern by BINARYINFERNO; N means that it was not; P indicates the pattern was partially identified.

Sample	Pattern	100	500	1000
`dhcp`	$(TL(V)^{[L]})*$	N	Y	Y
`mirai`	$Q(V^5)^{[Q]} \cdot Q(TL(V)^{[L]})^{[Q]}$	Y	Y	Y
`smb`	$Q(VV)^{[Q]} \cdot LL(V)^{[LL]}$	Y	Y	Y
`tutorial`	$(L(V)^{[L]})*$	P	Y	Y

We evaluated our serialization pattern inference approach for top-level samples (`dhcp`, `mirai`, `smb`, `tutorial`) with a ground-truth serialization pattern (`dhcp`, `tutorial`) or combination of patterns (`mirai`, `smb`) covered by our grammar. We summarize these results in Table IV. For samples of 1000 and 500 messages, BINARYINFERNO correctly inferred the exact serialization pattern used in each instance. For samples of 100 messages, BINARYINFERNO was unable to identify the pattern for `dhcp` and excluded the first 3 pattern bytes in `tutorial`, which we attribute to limited diversity in the sample data. The other six protocol samples did not have patterns in our grammar. BINARYINFERNO appropriately inferred a contiguous unknown region for the payloads in each case.

E. Testing on random data with no boundaries.

An analyst's time is important, and every false positive adds to their workload. We wanted to evaluate each tool's false positive rate on data which has no structure, and for which no boundaries should be inferred. To that end, we created 20 samples: 10 with variable-length messages and 10 with fixed-width messages, each containing 100 messages of uniformly distributed random byte values. We progressively increased the Shannon entropy across samples starting from 1 up to 8 bits. When given random data, BINARYINFERNO correctly inferred no field boundaries, resulting in an average FPR of 0.0, when run in big, little, and endian-agnostic modes. This result is critical, as it provides us (and our analyst users) confidence that BINARYINFERNO will not see false patterns in random data. Similarly, FIELDHUNTER had an FPR of 0.0, NETZOB a FPR of

TABLE V: Top-level protocol and encapsulated payload results; Bold indicates best result.

		Top-level Protocol Samples																				
		BI+			BI			AWRE			FIELDHUNTER			NEMESYSR			NETPLIER			NETZOB		
Sample	# Msgs	Pre.	Rec.	FPR	Pre.	Rec.	FPR	Pre.	Rec.	FPR	Pre.	Rec.	FPR	Pre.	Rec.	FPR	Pre.	Rec.	FPR	Pre.	Rec.	FPR
bgp	1000	**1.00**	**0.94**	**0.00**	**1.00**	**0.94**	**0.00**	0.51	0.06	0.005	**1.00**	**0.94**	**0.00**	0.06	0.50	0.36	0.22	0.53	0.09	0.00	0.00	0.02
dhcp	1000	**0.67**	0.75	0.02	0.66	**0.82**	0.03	0.07	0.01	**0.01**	*	*	*	0.49	0.36	0.02	0.18	0.31	0.08	-	-	-
dnp3	1000	0.61	0.31	**0.02**	0.61	0.31	**0.02**	0.00	0.00	0.07	*	*	*	0.34	0.54	0.13	0.23	**1.00**	0.42	**0.64**	0.40	0.03
mavlink	1000	**1.00**	0.67	**0.00**	0.72	0.84	0.03	0.50	0.17	0.03	**1.00**	0.33	**0.00**	0.39	0.78	0.11	0.17	**0.99**	0.43	**1.00**	0.50	**0.00**
mirai	1000	**1.00**	**1.00**	**0.00**	**1.00**	**1.00**	**0.00**	0.00	0.00	0.05	0.33	0.10	0.03	0.35	0.24	0.07	0.63	0.55	0.05	0.86	0.60	0.02
modbus	1000	**1.00**	0.40	**0.00**	0.60	0.60	0.05	0.00	0.00	0.09	0.76	0.60	0.02	0.54	0.59	0.06	0.23	**0.80**	0.34	0.60	0.60	0.05
ntp48	1000	**1.00**	0.70	**0.00**	0.26	0.50	0.16	0.00	0.00	0.04	*	*	*	0.39	0.61	0.11	0.22	**1.00**	0.42	0.38	0.31	0.06
smb	1000	**1.00**	0.59	**0.00**	0.73	0.73	0.02	0.01	0.002	0.04	*	*	*	0.17	0.25	0.07	0.39	**0.87**	0.08	0.41	0.45	0.04
smb2	1000	**1.00**	0.59	**0.00**	0.50	0.59	0.03	0.67	0.08	0.004	0.00	0.00	0.01	0.21	0.44	0.08	0.18	**0.99**	0.22	-	-	-
tutorial	1000	**1.00**	**1.00**	**0.00**	0.80	**1.00**	0.03	0.00	0.00	0.08	**1.00**	0.25	**0.00**	0.16	0.09	0.06	0.45	0.50	0.08	0.67	0.50	0.03
Average Performance		**0.93**	0.70	**0.005**	0.69	0.73	0.04	0.18	0.03	0.04	0.68	0.37	0.01	0.31	0.44	0.11	0.29	**0.75**	0.22	0.57	0.42	0.03
bgp	500	**1.00**	**0.94**	**0.00**	**1.00**	**0.94**	**0.00**	0.52	0.06	0.005	**1.00**	**0.94**	**0.00**	0.06	0.50	0.36	0.19	0.53	0.11	0.00	0.00	0.02
dhcp	500	**0.71**	0.67	0.02	0.70	**0.76**	0.02	0.07	0.005	**0.01**	*	*	*	0.49	0.37	0.02	0.18	0.24	0.07	-	-	-
dnp3	500	0.61	0.31	**0.02**	0.61	0.31	**0.02**	0.00	0.00	0.07	*	*	*	0.34	**0.54**	0.13	0.24	**1.00**	0.40	**0.64**	0.39	0.03
mavlink	500	**1.00**	0.67	**0.00**	0.72	0.84	0.03	0.50	0.17	0.03	**1.00**	0.33	**0.00**	0.39	0.78	0.11	0.16	**0.88**	0.41	**1.00**	0.50	**0.00**
mirai	500	**1.00**	**1.00**	**0.00**	**1.00**	**1.00**	**0.00**	0.00	0.00	0.05	0.33	0.10	0.03	0.35	0.24	0.07	0.51	0.60	0.09	0.86	0.60	0.02
modbus	500	**1.00**	0.40	**0.00**	0.60	0.60	0.05	0.00	0.00	0.09	0.76	0.60	0.02	0.55	0.59	0.07	0.26	**0.80**	0.30	0.60	0.60	0.05
ntp48	500	**1.00**	0.70	**0.00**	0.26	0.50	0.16	0.00	0.00	0.04	*	*	*	0.39	0.62	0.11	0.22	**1.00**	0.42	0.43	0.30	0.05
smb	500	**1.00**	0.59	**0.00**	0.70	0.66	0.02	0.01	0.002	0.04	*	*	*	0.20	0.28	0.07	0.31	**0.79**	0.10	0.38	0.46	0.05
smb2	500	**0.80**	0.59	0.01	0.47	0.59	0.03	0.68	0.08	**0.004**	0.00	0.00	0.01	0.22	0.45	0.08	0.18	**0.99**	0.23	-	-	-
tutorial	500	**1.00**	**1.00**	**0.00**	0.80	**1.00**	0.03	0.00	0.00	0.08	**1.00**	0.25	**0.00**	0.16	0.09	0.06	0.44	0.50	0.08	0.67	0.50	0.03
Average Performance		**0.91**	0.69	**0.005**	0.69	0.72	0.04	0.18	0.03	0.04	0.68	0.37	0.01	0.31	0.45	0.11	0.27	**0.73**	0.22	0.57	0.42	0.03
bgp	100	**1.00**	**0.89**	**0.00**	**1.00**	**0.89**	**0.00**	0.00	0.00	0.04	**1.00**	**0.89**	**0.00**	0.06	0.51	0.32	0.12	0.55	0.17	0.00	0.00	0.02
dhcp	100	0.05	0.14	0.15	0.05	0.14	0.15	0.07	0.01	**0.01**	0.00	0.00	**0.01**	**0.44**	0.34	0.02	0.37	**0.53**	0.05	-	-	-
dnp3	100	0.61	0.29	**0.02**	0.61	0.29	**0.02**	0.00	0.00	0.06	0.14	0.03	**0.02**	0.29	0.46	0.14	0.23	**1.00**	0.40	**0.64**	0.37	**0.02**
mavlink	100	**1.00**	0.67	**0.00**	0.51	0.51	0.04	0.00	0.00	0.05	*	*	*	0.40	0.80	0.11	0.15	**0.85**	0.42	**1.00**	0.50	**0.00**
mirai	100	**1.00**	**1.00**	**0.00**	**1.00**	**1.00**	**0.00**	0.00	0.00	0.05	0.00	0.00	0.01	0.34	0.24	0.07	0.44	**1.00**	0.19	0.86	0.60	0.01
modbus	100	**1.00**	0.40	**0.00**	0.60	0.60	0.05	0.00	0.00	0.09	0.77	0.60	0.02	0.54	0.58	0.07	0.25	**0.80**	0.32	0.60	0.60	0.05
ntp48	100	**0.75**	0.60	**0.02**	0.17	0.30	0.18	0.00	0.00	0.04	*	*	*	0.37	0.56	0.11	0.22	**1.00**	0.42	0.43	0.31	0.05
smb	100	**1.00**	0.59	**0.00**	0.78	0.73	0.01	0.004	0.002	0.05	*	*	*	0.19	0.30	0.08	0.22	**1.00**	0.21	0.33	0.42	0.05
smb2	100	**0.82**	0.71	0.01	0.45	0.71	0.05	0.69	0.08	**0.004**	*	*	*	0.24	0.48	0.08	0.18	**0.99**	0.24	0.20	0.76	0.15
tutorial	100	**1.00**	**1.00**	**0.00**	0.81	**1.00**	0.03	0.08	0.04	0.08	1.00	0.24	**0.00**	0.17	0.09	0.06	0.34	0.61	0.15	0.67	0.48	0.03
Average Performance		**0.82**	0.63	0.02	0.60	0.62	0.05	0.08	0.01	0.05	0.48	0.29	**0.01**	0.30	0.44	0.10	0.25	**0.83**	0.26	0.53	0.45	0.04

		Encapsulated Payload Samples																				
		BI+			BI			AWRE			FIELDHUNTER			NEMESYS			NETPLIER			NETZOB		
Sample	# Msgs	Pre.	Rec.	FPR	Pre.	Rec.	FPR	Pre.	Rec.	FPR	Pre.	Rec.	FPR	Pre.	Rec.	FPR	Pre.	Rec.	FPR	Pre.	Rec.	FPR
bgp01	22	**0.93**	0.77	**0.01**	**0.93**	0.77	**0.01**	0.00	0.00	0.02	0.51	0.23	0.04	0.29	0.51	0.24	0.48	**0.90**	0.18	0.46	0.53	0.12
bgp02	104	**1.00**	0.46	**0.00**	0.67	0.46	0.01	0.00	0.00	0.02	0.74	**0.67**	0.01	0.09	**0.67**	0.25	0.06	0.54	0.31	0.21	0.23	0.03
bgp04	1000	**1.00**	**1.00**	**0.00**	**1.00**	**1.00**	**0.00**	**1.00**	**1.00**	**0.00**	**1.00**	**1.00**	**0.00**	0.00	0.00	0.43	**1.00**	**1.00**	**0.00**	**1.00**	**1.00**	**0.00**
mavlink001	206	**1.00**	0.67	**0.00**	**1.00**	0.67	**0.00**	**1.00**	0.33	**0.00**	*	*	*	0.22	0.66	0.12	0.75	**1.00**	0.02	**1.00**	**1.00**	**0.00**
mavlink002	51	**1.00**	**1.00**	**0.00**	0.25	**1.00**	0.14	0.00	0.00	0.15	*	*	*	0.18	**1.00**	0.21	0.12	**1.00**	0.32	0.33	**1.00**	0.09
mavlink004	33	**1.00**	**0.50**	**0.00**	0.25	**0.50**	0.12	0.00	0.00	0.18	*	*	*	0.15	0.33	0.15	0.14	**0.50**	0.24	0.33	**0.50**	0.08
mavlink024	400	**1.00**	0.62	**0.00**	0.40	0.50	0.12	0.00	0.00	0.10	*	*	*	0.37	0.47	0.13	0.33	**0.88**	0.27	0.45	0.62	0.12
mavlink026	1000	*	*	*	0.00	0.00	0.05	0.00	0.00	0.09	*	*	*	-	-	-	0.00	0.00	0.07	**0.50**	**1.00**	**0.02**
mavlink030	1000	**1.00**	**1.00**	**0.00**	0.60	**1.00**	0.08	0.00	0.00	0.07	*	*	*	0.22	0.38	0.16	0.22	**1.00**	0.43	0.50	0.17	0.02
mavlink031	1000	**1.00**	**1.00**	**0.00**	0.78	**1.00**	0.04	0.00	0.00	0.06	*	*	*	0.23	0.38	0.15	0.23	**1.00**	0.43	0.50	0.14	0.02
mavlink032	1000	**1.00**	**1.00**	**0.00**	0.38	0.83	0.16	0.00	0.00	0.07	*	*	*	0.22	0.37	0.16	0.22	**1.00**	0.43	0.50	0.17	0.02
mavlink033	1000	**0.75**	0.75	**0.04**	0.46	0.75	0.15	0.25	0.12	0.10	*	*	*	0.45	0.63	0.13	0.31	**1.00**	0.38	0.51	0.63	0.10
mavlink042	371	**1.00**	**1.00**	**0.00**	0.00	0.00	0.33	0.00	0.00	0.55	**1.00**	**1.00**	**0.00**	-	-	-	0.00	0.00	0.33	0.00	0.00	0.33
mavlink046	5	**1.00**	**1.00**	**0.00**	0.00	0.00	0.33	0.00	0.00	0.56	**1.00**	**1.00**	**0.00**	-	-	-	0.00	0.00	0.33	0.00	0.00	0.33
mavlink076	93	*	*	*	*	*	*	0.00	0.00	0.06	*	*	*	0.06	**0.58**	0.26	**0.33**	0.50	**0.03**	**0.33**	0.50	**0.03**
mavlink083	385	**0.89**	0.89	**0.02**	0.58	0.78	0.08	0.00	0.00	0.08	*	*	*	0.26	0.35	0.14	0.25	**1.00**	0.42	0.60	0.33	0.03
mavlink085	377	**0.80**	0.73	**0.02**	0.37	0.64	0.13	0.00	0.00	0.13	*	*	*	0.30	0.43	0.12	0.25	**1.00**	0.37	0.25	0.18	0.07
mavlink087	247	**1.00**	**0.75**	**0.00**	0.50	**0.75**	0.03	0.00	0.00	0.04	*	*	*	0.37	0.66	0.05	0.15	0.50	0.11	0.29	0.50	0.05
mavlink111	410	**0.50**	**1.00**	**0.03**	0.33	**1.00**	0.07	0.00	0.00	0.11	*	*	*	0.10	**1.00**	0.30	0.17	**1.00**	0.17	0.33	**1.00**	0.07
mavlink140	383	**1.00**	0.57	**0.00**	0.50	0.57	0.05	0.00	0.00	0.05	*	*	*	0.27	0.39	0.10	0.26	**0.86**	0.23	0.33	0.43	0.08
mavlink141	388	**0.67**	0.80	**0.03**	0.33	0.60	0.10	0.00	0.00	0.06	*	*	*	0.28	0.61	0.13	0.24	**1.00**	0.28	0.33	0.40	0.07
mavlink147	207	**1.00**	0.40	**0.00**	**1.00**	0.40	**0.00**	0.00	0.00	0.05	*	*	*	0.18	0.36	0.12	0.62	**1.00**	0.05	**1.00**	0.60	**0.00**
mavlink230	39	**1.00**	**1.00**	**0.00**	0.89	**1.00**	0.01	0.00	0.00	0.05	*	*	*	0.29	0.42	0.11	0.26	**1.00**	0.31	0.45	0.62	0.08
mavlink241	30	**1.00**	**1.00**	**0.00**	0.80	**1.00**	0.02	0.14	0.25	0.16	*	*	*	0.29	0.54	0.09	0.19	0.75	0.22	0.50	0.50	0.03
mavlink242	14	**1.00**	0.44	**0.00**	**1.00**	0.44	**0.00**	*	*	*	*	*	*	0.53	0.77	0.06	0.31	**0.89**	0.19	0.55	0.67	0.05
mavlink245	191	*	*	*	*	*	*	**1.00**	**1.00**	**0.00**	*	*	*	-	-	-	**1.00**	**1.00**	**0.00**	**1.00**	**1.00**	**0.00**
Average Performance		**0.94**	**0.80**	**0.01**	0.54	0.65	0.08	0.14	0.11	0.11	0.85	0.78	**0.01**	0.24	0.52	0.17	0.30	0.78	0.24	0.47	0.53	0.07

Results with timeouts (-) or where a tool found no fields (*) are excluded from averages.

0.01, and AWRE a FPR of 0.02, while FIELDHUNTER inferred no field boundaries for any random sample. NETZOB's false positives increased proportionally with the number of repeated byte values in a sample. NEMESYS had an average FPR of 0.40, and NETPLIER an average FPR of 0.45.

F. Testing the Entropy Field-Boundary Detector

We wanted to evaluate BINARYINFERNO's field boundary detector to understand how it would perform if semantic detectors are unable to recognize sample data, or simply do not exist for a specific field type. The motivation here is to understand how much confidence an analyst should have in BINARYINFERNO when using only the heuristic method described in Section VI for identifying field boundaries.

We evaluated BINARYINFERNO's entropy boundary detector in isolation on our top-level samples of 1000 messages and our encapsulated payload samples. For each sample we inferred field boundaries giving BINARYINFERNO first the correct ground-truth endianess for the sample, and then inferring again using opposite endianness. When given the correct endianess as a prior, BINARYINFERNO had an average precision of 0.88 and an average recall of 0.35 for top-level samples of 1000 messages, and an average precision of 0.96 and average recall of 0.63 for encapsulated payloads as shown in Appendix F Table VII. When given the correct endianess, BINARYINFERNO produced strong results using only this detector. We attribute the difference in recall between top-level and payloads to the larger proportion of fixed-length formats in the payload samples, and consequently more fields which are byte-aligned across messages.

G. Measuring execution times.

To characterize how long it takes BINARYINFERNO and related tools to perform field inference, we recorded execution times for our top-level evaluation as summarized in Figure 7. AWRE, FIELDHUNTER, and NEMESYS completed each of 30 samples (10 formats at 3 different sample sizes) in less than 60 seconds, while BINARYINFERNO completed 22 out of 30 in 60 seconds, and NETPLIER completed 21. NETZOB's multiple-sequence alignment approach timed out after 60 minutes on two samples: `dhcp`, and `smb`. Unsurprisingly, samples with longer messages required more time for BINARYINFERNO, NETPLIER, and NETZOB. Longer messages typically have more valid offsets from which BINARYINFERNO can initiate pattern-search, increasing execution time. We note that BINARYINFERNO's execution time on `dhcp` was notably long at almost 1500 seconds. This sample contained large zero-valued slices in the header fields. These slices caused our pattern-based detector to consider numerous spurious serialization patterns with zeroes supplying lengths or quantities. As a result, on `dhcp`, our pattern search algorithm explored many consistent patterns caused by the many zero slices. A possible solution to this issue would be heuristics limiting the number of zero-valued patterns allowed when performing inference. Finally, we note that all of these times are short compared to the time an analyst takes to reverse-engineer a format by hand.

H. Discussion

BINARYINFERNO's performance inferring serialization patterns is notable, as it was able to correctly infer the exact

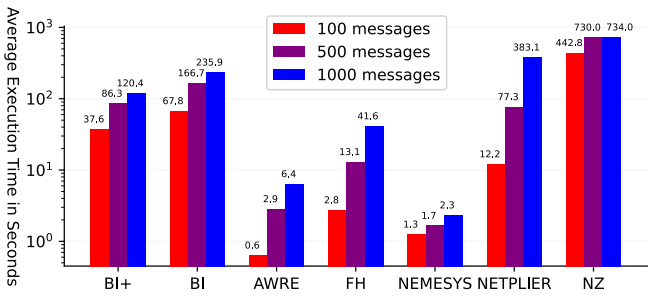

Fig. 7: Log scale plot of average execution times in seconds on top-level samples of 100, 500, and 1000 messages for BINARYINFERNO (BI), BINARYINFERNO with priors (BI+), AWRE, FIELDHUNTER (FH) NEMESYS, NETPLIER, and NETZOB (NZ). Shorter execution times are better.

serialization format for four of the top-level samples. By exactly inferring the serialization format, it can produce a parser for future messages. BINARYINFERNO's approach means that not only are field boundaries correctly identified, but the semantic relationship describing each portion of a variable-length message is also uncovered. This approach contrasts with tools such as NEMESYS, NETPLIER, and NETZOB, which only identify field boundaries and require further manual inspection by an analyst to uncover the semantic relationship in the data. AWRE and FIELDHUNTER offer no support for such patterns.

A key advantage of BINARYINFERNO's approach is that when it cannot explain a portion of the format, such as when given random-valued data or a pattern outside its serialization grammar, it leaves it untouched. This result allows analysts to focus their attention on things which cannot be explained automatically, while letting BINARYINFERNO identify those fields which can. For example, the `bgp02`, `smb2`, and `dnp3` ground-truth formats all serialize variable-length data according to patterns outside our serialization pattern grammar and as a result were left as unexplained variable-length fields by BINARYINFERNO. The `bgp02` format uses a type-length-value (TLV) format where the width of the length field is determined by the specific bits in the value of each type field, and the `smb2` format encodes the length of some portions of each message using only 15 of the 16 bits in a 2-byte length field. The `dnp3` format interleaves a 2-byte CRC for every 16 bytes of payload, which is also outside our grammar. Presently, BINARYINFERNO is only capable of inferring byte-aligned fields and variable-length regions covered by our grammar. The `dnp3` sample was further challenging for BINARYINFERNO due to portions of the interleaved CRC codes appeared as spurious floats.

In some cases, tools were unable to identify any field boundaries for a sample: FIELDHUNTER (21 payloads, 4 top-level), BINARYINFERNO (3 payloads), and AWRE (1 payload). For instances where BINARYINFERNO was unable to identify field boundaries, the consistent cause was a lack of diversity in the samples, with some samples taking on only 2 or 4 distinct message values. FIELDHUNTER failed to identify any fields for the majority of the evaluation samples. Finally, while more data is generally helpful, in the case of `dhcp`, we observed 2 spurious float fields which were inferred from the 1000 message sample, but not from the 500 message sample,

resulting in lower precision. Similarly, with `smb2` additional data allowed spurious fields to be inferred. We discuss related problems and possible solutions in Section XI.

X. Related Work

Protocol reverse engineering is a widely studied problem well summarized in recent survey papers by Narayan et al. [50], Duchene et al. [31], and Kleber et al. [39]. Reverse engineering approaches can be characterized by which aspect of the problem they attempt to automate: message clustering, format inference, and/or state machine inference. We focus on methods that take a static network trace as input and produce an inferred format as output because such work is the most closely related to ours. Alternative approaches, including static analysis [58], [59], binary instrumentation [20], [21], [24], [27], [44], [45], [52], [63] and fuzzing [18], [35], [66], all rely on artifacts that are out of scope for our work.

Bossert et al.'s Netzob [19] is a collection of Python libraries that help an analyst interactively reverse engineer an unknown protocol in a Python interpreter. Netzob implements common algorithms used as part of general protocol reverse engineering, including sequence alignment and state machine inference, but relies on an analyst to decide what operations to apply to a sample and how to interpret the results. Netzob is one of several tools [42], [60] that use the Needleman-Wunsch algorithm [51], or similar multiple sequence alignment (MSA) [32], [33] techniques to infer field boundaries. These techniques depend on data commonalities across messages as the basis for inferring field boundaries, and do not infer semantics. BinaryInferno instead uses serialization pattern search to look for semantically meaningful explanations without relying on data commonalities. Ye et al.'s Netplier [64] performs message clustering and format inference by pairing MSA with probabilistic field inference. Netplier uses MSA to infer field boundaries; except for fields that are keywords (message type fields), the resulting alignments require further analysis to discover field semantics. In contrast, BinaryInferno performs field inference using an ensemble of detectors, producing semantic descriptions directly. Kleber et al.'s Nemesys [38] uses a measure of the relative changes between the bits in adjacent message bytes, termed *bit congruence*, to infer field boundaries in individual binary messages. BinaryInferno instead uses Shannon entropy to solve this problem by comparing bytes across multiple messages at a specific offset. Pohl et al.'s Automatic Wireless Protocol Reverse Engineering (Awre) [53] is designed to assist an analyst with reverse engineering the format of wireless radio protocols and is a sub-component of the Universal Radio Hacker security tool [3]. Awre runs each ensemble member over a collection of messages and chooses the first instance of each field type it detects. In contrast, BinaryInferno uses its integration algorithm to choose inferred fields for inclusion so as to maximize the amount of data described. Awre cannot identify complicated serialization patterns as BinaryInferno does. Bermudez et al.'s FieldHunter [17] performs field inference using a combination of semantic types and analyst supplied clear-text values expected to be in the network traffic, such as values captured from an execution trace. FieldHunter looks for correlations between regions of bytes and known clear-text values or facts such as message length to identify fields. Cui et al.'s Discoverer [26] is a protocol classification tool which uses tokenization, recursive clustering, and merging to perform both message clustering and format inference simultaneously. Discoverer assumes the analyst supplies knowledge of field delimiters, unlike other approaches. For binary protocols, Discoverer produces no semantics, describing fields only as constant or variable-valued.

Similarly to BinaryInferno, Awre, and FieldHunter are semantic approaches by Ládi et al. [41] and Fisher et al. [34].

[3] https://github.com/jopohl/urh

XI. Limitations and Future Work

Our approach has four major limitations. First, our approach only performs inference for offset-aligned fields and payload regions common across all messages in the sample, leaving union-types or optional fields untouched. In future work, we plan to expand our approach to automatically identify message type (keyword) fields to handle these latter cases. We hypothesize that we can use our general technique of partitioning a mixed collection of messages into groups with distinct formats, similar to Discoverer's recursive clustering [26] and Netplier's keyword identification approach [64]. Our evaluation on `mavlink` and `bgp` suggests this direction has promise. We believe this approach will also allow us to better handle field-level union types exhibiting rich heterogeneity where tag fields function as inline keywords. Second, our pattern detector is limited to patterns described in Grammar 2. An example of a serialization pattern not currently covered is DNS, where a length or quantity is some number D bytes from the value it describes. We plan to address such patterns in by automatically identifying long-distance dependencies through an extension of our star pattern inference technique. Third, our approach operates at the byte level, but some data formats describe data at the bit level. We plan to adapt our field-boundary and length-field techniques to handle this case by adjusting our minimum field size. Fourth, our approach is not presently tolerant to noise or corruption in the sample. We believe that our mechanism for computing weights can handle noise, given extensions to our implementation, but more research is needed.

As our ensemble of detectors increases, so does the potential for overlap: two detectors inferring competing descriptions for the same bytes. We believe that our integration algorithm can resolve such cases by scaling the weights used in our graph using the selectiveness of each detector, with more selective detectors awarded a higher weight. A related question occurs when multiple serialization patterns are inferred to describe the same region of data. We believe by deserializing and recursively performing inference we will be able to better rank serialization patterns. Similarly, we believe this approach can improve our iterative-deepening depth first search by prioritizing patterns in the same manner.

XII. Conclusion

In this paper we presented BinaryInferno, our automatic approach to reverse engineering binary message formats. BinaryInferno identified field boundaries in top-level protocols with an average precision of 0.69, average recall of 0.73, and a false positive rate of 0.04, significantly outperforming five state-of-the-art tools for field inference.

Acknowledgments

This material is based upon work partly supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-19-C-0073. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for Public Release, Distribution Unlimited.

Availability

We will release BinaryInferno open source accompanied by all datasets used in our evaluation.

References

[1] 4SICS. https://www.netresec.com/?page=PCAP4SICS.

[2] BinaryInferno. https://binaryinferno.net.

[3] jMAVSim. https://github.com/PX4/jMAVSim.

[4] mp-bgp-capture. https://weberblog.net/mp-bgp-capture.

[5] NetPlier. https://github.com/netplier-tool/NetPlier.

[6] PXE.PCAP. https://www.cloudshark.org/captures/1fd97aede26b.

[7] Reverse Engineering Network Protocols. https://jhalon.github.io/reverse-engineering-protocols.

[8] SMIA2011. http://download.netresec.com/pcap/smia-2011.

[9] tcpdump. https://www.tcpdump.org.

[10] The Ultimate PCAP. https://weberblog.net/the-ultimate-pcap.

[11] UCI Machine Learning Repository. https://archive.ics.uci.edu.

[12] Wireshark. https://wireshark.org.

[13] IEEE Standard for Electric Power Systems Communications-Distributed Network Protocol (DNP3). *IEEE Std 1815-2012 (Revision of IEEE Std 1815-2010)*, pages 1–821, 2012.

[14] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[15] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the Mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110, 2017.

[16] Sukhrob Atoev, Ki-Ryong Kwon, Suk-Hwan Lee, and Kwang-Seok Moon. Data analysis of the mavlink communication protocol. In *2017 International Conference on Information Science and Communications Technologies (ICISCT)*, pages 1–3. IEEE, 2017.

[17] Ignacio Bermudez, Alok Tongaonkar, Marios Iliofotou, Marco Mellia, and Maurizio M Munafo. Automatic protocol field inference for deeper protocol understanding. In *2015 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2015.

[18] Bernhards Blumbergs and Risto Vaarandi. Bbuzz: A bit-aware fuzzing framework for network protocol systematic reverse engineering and analysis. In *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*, pages 707–712. IEEE, 2017.

[19] Georges Bossert and Frederic Guihery. Security evaluation of communication protocols in common criteria. In *Proc of IEEE International Conference on Communications*, 2012.

[20] Georges Bossert, Frédéric Guihéry, and Guillaume Hiet. Towards automated protocol reverse engineering using semantic information. In *Proceedings of the 9th ACM Aymposium on Information, Computer and Communications Security*, pages 51–62, 2014.

[21] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 317–329, 2007.

[22] Vinton Cerf and Robert Kahn. A protocol for packet network intercommunication. *IEEE Transactions on communications*, 22(5):637–648, 1974.

[23] Jared Chandler, Kathleen Fisher, Erin Chapman, Eric Davis, and Adam 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.

[24] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *USENIX Security Symposium*, volume 139, 2011.

[25] Chia Yuan Cho, Eui Chul Richard Shin, Dawn Song, et al. Inference and Analysis of Formal Models of Botnet Command and Control Protocols. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 426–439. ACM, 2010.

[26] Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*, pages 1–14, 2007.

[27] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications security*, pages 391–402, 2008.

[28] Lorenzo De Carli, Ruben Torres, Gaspar Modelo-Howard, Alok Tongaonkar, and Somesh Jha. Botnet Protocol Inference in the Presence of Encrypted Traffic. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[29] Ralph Droms. Dynamic Host Configuration Protocol. RFC 1541, October 1993.

[30] Olivier Dubuisson. *ASN. 1 communication between heterogeneous systems*. Morgan Kaufmann, 2000.

[31] Julien Duchene, Colas Le Guernic, Eric Alata, Vincent Nicomette, and Mohamed Kaâniche. State of the art of network protocol reverse engineering tools. *Journal of Computer Virology and Hacking Techniques*, 14(1):53–68, 2018.

[32] Robert C Edgar and Serafim Batzoglou. Multiple sequence alignment. *Current Opinion in Structural Biology*, 16(3):368–373, 2006.

[33] Da-Fei Feng and Russell F Doolittle. Progressive sequence alignment as a prerequisitetto correct phylogenetic trees. *Journal of Molecular Evolution*, 25(4):351–360, 1987.

[34] Kathleen Fisher, David Walker, Kenny Q Zhu, and Peter White. From dirt to shovels: Fully automatic tool generation from ad hoc data. In *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'08*, pages 421–434, 2008.

[35] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*, pages 330–347. Springer, 2015.

[36] Obinna Igbe, Ihab Darwish, and Tarek Saadawi. Deterministic dendritic cell algorithm application to smart grid cyber-attack detection. 06 2017.

[37] X ITU-T. 690: Itu-t recommendation x. 690 (1997) information technology-asn. 1 encoding rules: Specification of basic encoding rules (ber). *Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*.

[38] Stephan Kleber, Henning Kopp, and Frank Kargl. NEMESYS: Network message syntax reverse engineering by analysis of the intrinsic structure of individual messages. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[39] Stephan Kleber, Lisa Maile, and Frank Kargl. Survey of protocol reverse engineering algorithms: Decomposition of tools for static traffic analysis. *IEEE Communications Surveys & Tutorials*, 2018, 2018.

[40] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[41] Gergő Ládi, Levente Buttyán, and Tamás Holczer. Message format and field semantics inference for binary protocols using recorded network traffic. In *2018 26th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–6. IEEE, 2018.

[42] Corrado Leita, Ken Mermoud, and Marc Dacier. Scriptgen: An automated script generation tool for honeyd. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 12–pp. IEEE, 2005.

[43] Wentian Li. Mutual information functions versus correlation functions. *Journal of Statistical Physics*, 60(5-6):823–837, 1990.

[44] Junghee Lim, Thomas Reps, and Ben Liblit. Extracting output formats from executables. In *2006 13th Working Conference on Reverse Engineering*, pages 167–178. IEEE, 2006.

[45] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *NDSS*, volume 8, pages 1–15. Citeseer, 2008.

[46] Jim Martin, Jack Burbank, William Kasch, and Professor David L. Mills. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905, June 2010.

[47] Microsoft Corporation. Server message block (smb) protocol, 2022. [Online; accessed 20-March-2022].

[48] Microsoft Corporation. Server message block (smb) protocol versions 2 and 3, 2022. [Online; accessed 20-March-2022].

[49] Modbus Organization, Inc. Modbus application protocol specification v1.1b3, 2022. [Online; accessed 20-March-2022].

[50] John Narayan, Sandeep K Shukla, and T Charles Clancy. A survey of automatic protocol reverse engineering tools. *ACM Computing Surveys (CSUR)*, 48(3):1–26, 2015.

[51] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[52] James Newsome, David Brumley, Jason Franklin, and Dawn Song. Replayer: Automatic protocol replay by binary analysis. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 311–321, 2006.

[53] Johannes Pohl and Andreas Noack. Automatic wireless protocol reverse engineering. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019.

[54] Jon Postel. User datagram protocol. Technical report, 1980.

[55] Yakov Rekhter, Susan Hares, and Tony Li. A Border Gateway Protocol 4 (BGP-4). RFC 4271, January 2006.

[56] Joyce K. Reynolds and Dr. Jon Postel. Assigned Numbers. RFC 1700, October 1994.

[57] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, July 1948.

[58] Octavian Udrea, Cristian Lumezanu, and Jeffrey S Foster. Rule-based static analysis of network protocol implementations. *Information and Computation*, 206(2-4):130–157, 2008.

[59] Sharath K Udupa, Saumya K Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering (WCRE'05)*, pages 10–pp. IEEE, 2005.

[60] Yipeng Wang, Xiaochun Yun, M Zubair Shafiq, Liyan Wang, Alex X Liu, Zhibin Zhang, Danfeng Yao, Yongzheng Zhang, and Li Guo. A semantics aware approach to automated reverse engineering unknown protocols. In *2012 20th IEEE International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE, 2012.

[61] Wikipedia contributors. Mutual information — Wikipedia, the free encyclopedia, 2021. [Online; accessed 20-July-2021].

[62] David H Wolpert and David R Wolf. Estimating functions of probability distributions from a finite set of samples. *Physical Review E*, 52(6):6841, 1995.

[63] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, and Scuola Superiore S Anna. Automatic network protocol analysis. In *NDSS*, volume 8, pages 1–14. Citeseer, 2008.

[64] Yapeng Ye, Zhuo Zhang, Fei Wang, Xiangyu Zhang, and Dongyan Xu. NetPlier: probabilistic network protocol reverse engineering from message traces. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS'21)*, 2021.

[65] Jin Y Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.

[66] Hui Zhao, Zhihui Li, Hansheng Wei, Jianqi Shi, and Yanhong Huang. Seqfuzzer: An industrial protocol fuzzing framework from a deep learning perspective. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 59–67. IEEE, 2019.

Appendix A
L-Ratio cut-off points for Float Detector

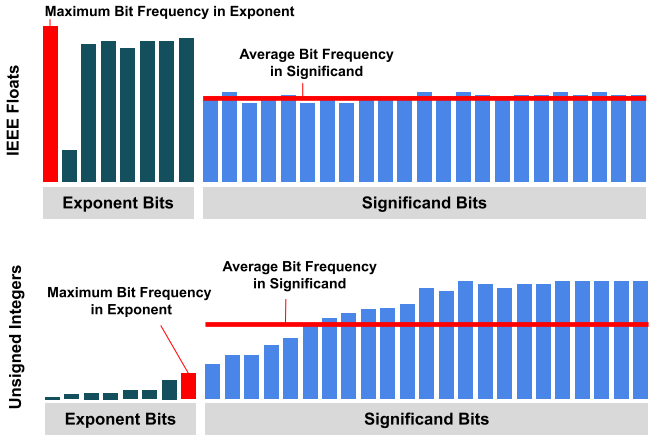

Fig. 8: Example histogram of set bits by position across a sample of floats illustrating the L-Shape feature vs a sample of unsigned integers illustrating no L-Shape feature.

To determine appropriate cut-off points for the L-Ratio, we calculated the L-Ratio for 4-byte float data slices drawn from values in five machine learning data example datasets [11] [4] [5] [6] [7] [8], 4-byte random data slices, and 4-byte integer data slices. We also calculated the ratio for 4-byte slices constructed as a sliding window across combinations of the previous three slice types. We observed that the majority of non-float slices had L-Ratios less than .42 or greater than .55 as illustrated in Figure 9. As a result, we selected those cutoff points for use in our float detector. We believe this approach will generalize to double-precision floats as the relationship between exponent and significand is unchanged.

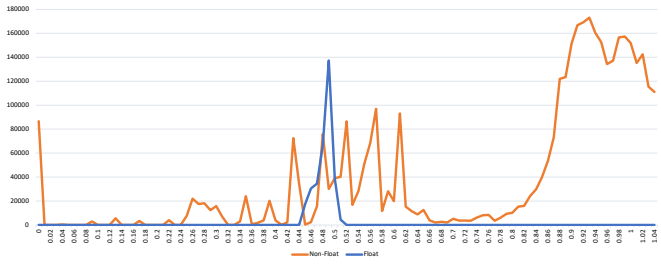

Fig. 9: Visualization of L-Ratio feature for samples drawn from Floats (blue) and Non-Floats (orange). The X-axis represents L-Ratio. The Y-axis represents the frequency of floats and non-floats that exhibit a particular L-Ratio value.

[4] https://archive.ics.uci.edu/ml/machine-learning-databases/housing
[5] https://archive.ics.uci.edu/ml/machine-learning-databases/ionosphere
[6] https://archive.ics.uci.edu/ml/datasets/QSAR+aquatic+toxicity
[7] https://archive.ics.uci.edu/ml/machine-learning-databases/00514
[8] https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin

Appendix B
Could random data be mistaken for a length field?

TABLE VI: Probability of random length field in collection with n possible 1-byte slices over m messages.

	n = 1	n = 10	n = 100	n = 1000
1 message	0.39%	3.84%	32.39%	98.00%
2 messages	≃0.00%	0.02%	0.15%	1.51%
3 messages	≃0.00%	≃0.00%	≃0.00%	0.01%
4 messages	≃0.00%	≃0.00%	≃0.00%	≃0.00%

Our approach leverages multiple messages from the same format to perform two functions. First: to provide evidence for field types, and second: to restrict the possible inferred fields through contradiction. One concern, then, is when there is enough for the first function, but not enough for the second. For length fields, for example: how often is it that a random slice of the data happens to be equal to the lengths of the messages purely by coincidence? In Equation 7, we calculate the probability that a 1-byte slice is randomly consistent with the message lengths.

$$P(n,m,d) = \sum_{k=1}^{n} \binom{n}{k} \left(\frac{1}{d}\right)^{mk} \left(\frac{d^m - 1}{d^m}\right)^{n-k} \tag{7}$$

This probability function is parameterized by the number of slices (n) we can take over a collection, the number of messages (m) in the the collection, and the number of values (d) each byte can take on; in the case of a 1-byte length field: 256. We sum the probabilities that (k) slices equal the lengths of the messages from 1 to n, where all slices encode the lengths. For each choice of k we calculate the number of combinations of slices as $\binom{n}{k}$. We calculate the probability of each slice taking on the values of the lengths as $(\frac{1}{d})^{mk}$. This term states that each of the m bytes in the k slices has to take on a specific value. We account for the remaining $n-k$ slices varying by at least one byte in the term $(\frac{d^m-1}{d^m})^{n-k}$. Intuitively, as we add more messages to our collection, we lower our risk of inferring a length field where none exists.

Appendix C
Fuzzy Lengths

Not all formats directly encode the length of messages, however. Some formats encode message lengths in terms of multi-byte units, or add error correcting codes at regular intervals and omit the error correcting bytes from the length value (this is the case with the `dnp3` protocol, for example, which is used in SCADA systems). Stated mathematically, the length fields in these formats do not directly encode the overall message length, but maintain a bijective relationship between the field value and overall message length. We call such fields *fuzzy lengths*. To address these cases, our second approach for identifying message lengths leverages normalized mutual information [43], [61], [62] (NMI) to identify slices that have high explanatory power with respect to the variation in message lengths. NMI describes the mutual dependence between two variables, and thus the amount of information that may be obtained about one such variable by observing the other. For our task, NMI measures how much a slice of bytes explains the lengths of the corresponding messages.

We infer a fuzzy length field when the slice values have a NMI value of 1.0 with message lengths, when slice values show a correlation of $\geq .95$ with message lengths, and when each slice value is less than or equal to the corresponding message length. The NMI condition ensures that the relationship between the slice values and message lengths is bijective. The correlation condition ensures an approximately linear relationship between the field values and message lengths. The final condition ensures each field value cannot describe a message length in excess of the actual message data. A strict length field will also satisfy our conditions for inferring a fuzzy length field. In such cases, the detector chooses to infer only the strict length field because it is more informative.

Appendix D
Star Patterns

To enable this calculation, we extend the memoization algorithm described earlier. Specifically, we calculate the reachable offsets (*reachable*) by creating a second set of memos: **STARMEMO**. Later, we will use the entries in **STARMEMO** to update **MEMO** so that we can treat star patterns just like the other serialization patterns in our search. We populate each entry **STARMEMO**[*pattern*, *message*, *offset*] with two values: the new offset (*newoffset*) we reach by interpreting the pattern from our current position and the set of all subsequent offsets (*subsequent*) we could reach from *newoffset*. We combine these two and update **STARMEMO** in the following manner:

$$\textit{newoffset} \leftarrow \textbf{MEMOS}[\textit{pattern}, \textit{message}, \textit{offset}]$$

$$\textit{subsequent} \leftarrow \textbf{STARMEMO}[\textit{pattern}, \textit{message}, \textit{newoffset}]$$

$$\textit{reachable} \leftarrow \{\textit{newoffset}\} \bigcup \textit{subsequent}$$

$$\textbf{STARMEMO}[\textit{pattern}, \textit{message}, \textit{offset}] \leftarrow \textit{reachable}$$

To construct **STARMEMO** efficiently, we work backwards from the last offset in each message to the first. By ordering our construction in this way, we ensure that the entries we need to populate *subsequent* will be memoized prior to their use. In cases where a pattern is uninterpretable at an offset and **MEMOS** returns a nil value, we set *reachable* to $\emptyset$.

To determine whether a pattern can derive a single consistent offset across all messages, we convert the *reachable* offsets (relative to the start of the message) into offsets relative to the end of the message. This computation is straightforward as we know the length of each message. We indicate these end-relative offsets as $\textit{offset}_{END}$, $\textit{reachable}_{END}$ and $\textbf{STARMEMO}[\textit{pattern}, \textit{message}_i, \textit{offset}]_{END}$.

Given the memos $\textbf{STARMEMO}_{END}$, we infer a star pattern over a collection of messages at an offset when $\bigcap_{i=0}^{|messages|} \textbf{STARMEMO}[\textit{pattern}, \textit{message}_i, \textit{offset}]_{END}$ is a non-empty set. Intuitively this set contains $\textit{offset}_{END}$ values reachable for every message in the collection. Out of this set, we choose the smallest element ($\textit{minoffset}_{END}$) as the parameter S in $\langle \textit{PATTERN} \rangle * \texttt{BYTE}^S$ for our star pattern on the principle of explaining as much of the sample data as possible.

Finally, we update **MEMOS** so that our search can treat star patterns identically to the patterns already stored. We do

this by creating appropriate entries for the new star pattern (*pattern**) in **MEMOS** for each *message* as follows.

MEMOS[*pattern**, *message*, *offset*] ← *minoffset*

In the example below we show the calculation of entries in $\textbf{STARMEMO}_{END}$ for the length-value pattern $L(V)^{[L]}$ starting at byte 7 in each message of our running example.

```
Msg1 0x054150504c45
Msg2 0x064f52414e4745
Msg3 0x04504c554d0450454152

Msg1 [5, 65, 80, 80, 76, 69]
Msg2 [6, 79, 82, 65, 78, 71, 69]
Msg3 [4, 80, 76, 85, 77, 4, 80, 69, 65, 82]

STARMEMO1 [ {0} ,∅, ∅, ∅, ∅, ∅]
STARMEMO2 [ {0} ,∅, ∅, ∅, ∅, ∅, ∅]
STARMEMO3 [{5,0},∅, ∅, ∅, ∅, {0}, ∅, ∅, ∅, ∅]
```

Calculating $\bigcap_{i=0}^{|messages|} \textbf{STARMEMO}[L(V)^{[L]}, message_i, 7]_{END}$ yields $\{0\} \bigcap \{0\} \bigcap \{5,0\} = \{0\}$ indicating that the pattern application is valid at offset 7. Alternatively, consider $\bigcap_{i=0}^{|messages|} \textbf{STARMEMO}[L(V)^{[L]}, message_i, 12]_{END}$. This gives $\emptyset \bigcap \emptyset \bigcap \{0\} = \emptyset$ indicating that the pattern cannot be applied at offset 12. Similar calculations at offsets 8 through 11 will also yield $\emptyset$.

Consider an alternative set of messages with two bytes added to end of each message.

```
Msg1 0x054150504c450145
Msg2 0x064f52414e47454f45
Msg3 0x04504c554d04504541525045

Msg1 [5, 65, 80, 80, 76, 69, 1, 69]
Msg2 [6, 79, 82, 65, 78, 71, 69, 79, 69]
Msg3 [4, 80, 76, 85, 77, 4, 80, 69, 65, 82, 80, 69]

STARMEMO1 [{2,0},∅, ∅, ∅, {0}, ∅]
STARMEMO2 [ {2} ,∅, ∅, ∅, ∅, ∅, ∅]
STARMEMO3 [{7,2},∅, ∅, ∅, ∅, {2}, ∅, ∅, ∅, ∅]
```

Our calculation at offset 7 now becomes $\{2,0\} \bigcap \{2\} \bigcap \{7,2\} = \{2\}$ indicating that the pattern application is valid at offset 7 and that there are 2 bytes remaining after the pattern application in every message. Note how our approach ignores the spurious pattern instance at the end of the first message to infer a description consistent across all messages.

Appendix E
Removing Padding

To automatically identify and remove padding, we search over candidate padding byte-values. For each candidate value, we remove all trailing instances of that value. When all messages are left with a single consistent stop byte (0xFF in the case of `dhcp`, we infer the candidate padding byte-value is correct, and return the messages with this padding removed. As part of our evaluation, we used this method of automatic padding detection and removal on all samples. Padding was detected and removed on for the `dhcp` sample. While `dhcp` is not strict in how padding is applied to the end of messages, other formats can be. For cases where padding is not explicitly accounted for by a length field or serialization pattern, another approach to identifying padding it is to observe that every message should end on a word boundary. By subtracting the minimum message length from the length of each message in a sample and factoring the resulting differences, candidate word sizes can be estimated and padding stripped to word boundaries.

Appendix F
Additional Tables

TABLE VII: Performance of BinaryInferno's field-boundary detector in isolation. Correct Endianess shows the performance of BinaryInferno when given format endianess as a prior. Incorrect Endianess show performance when given the opposite.

Sample	Endianess	Precision	Recall	FPR	F1
Top-level	Correct	0.88	0.35	0.00	0.47
Payload	Correct	0.96	0.63	0.00	0.68
Top-level	Incorrect	0.38	0.14	0.04	0.18
Payload	Incorrect	0.26	0.15	0.11	0.06

TABLE VIII: Descriptive statistics of samples.

Top-level Protocol Samples							
Sample	Endian	# Msgs.	Entropy	Length	Min.	Avg.	Max.
bgp	Big	1000	2.25	Variable	19	23	112
dhcp	Big	1000	2.25	Variable	241	289	408
dnp3	Little	1000	4.09	Variable	15	44	292
mavlink	Little	1000	6.67	Variable	10	36	59
mirai	Big	1000	5.96	Variable	22	38	52
modbus	Big	1000	3.68	Variable	10	22	239
ntp48	Big	1000	6.39	Fixed	48	48	48
smb	Little	1000	4.23	Variable	35	132	1408
smb2	Little	1000	3.07	Variable	68	142	800
tutorial	Big	1000	5.09	Variable	14	35	72
bgp01	Big	22	3.28	Variable	53	55	61
Encapsulated Payload Samples							
Sample	Endian	# Msgs.	Entropy	Length	Min.	Avg.	Max.
bgp02	Big	104	3.74	Variable	23	61	112
bgp04	Big	1000	1.12	Fixed	19	19	19
mavlink001	Little	206	2.25	Fixed	31	31	31
mavlink002	Little	51	5.82	Fixed	12	12	12
mavlink004	Little	33	3.68	Fixed	14	14	14
mavlink024	Little	400	5.30	Fixed	30	30	30
mavlink026	Little	1000	1.70	Fixed	22	22	22
mavlink030	Little	1000	7.47	Fixed	28	28	28
mavlink031	Little	1000	7.48	Fixed	32	32	32
mavlink032	Little	1000	7.49	Fixed	28	28	28
mavlink033	Little	1000	5.97	Fixed	28	28	28
mavlink042	Little	371	2.10	Fixed	2	2	2
mavlink046	Little	5	2.23	Fixed	2	2	2
mavlink076	Little	93	0.71	Fixed	33	33	33
mavlink083	Little	385	6.91	Fixed	37	37	37
mavlink085	Little	377	6.55	Fixed	51	51	51
mavlink087	Little	247	2.30	Fixed	51	51	51
mavlink111	Little	410	3.41	Fixed	16	16	16
mavlink140	Little	383	4.85	Fixed	41	41	41
mavlink141	Little	388	5.80	Fixed	32	32	32
mavlink147	Little	207	2.37	Fixed	36	36	36
mavlink230	Little	39	5.89	Fixed	42	42	42
mavlink241	Little	30	4.59	Fixed	32	32	32
mavlink242	Little	14	3.95	Fixed	52	52	52