Lifting Network Protocol Implementation to Precise Format Specification with Security Applications

Qingkai Shi
Purdue University
USA
shi553@purdue.edu

Junyang Shao
Purdue University
USA
shao156@purdue.edu

Yapeng Ye
Purdue University
USA
ye203@purdue.edu

Mingwei Zheng
Purdue University
USA
zheng618@purdue.edu

Xiangyu Zhang
Purdue University
USA
xyzhang@cs.purdue.edu

ABSTRACT

Inferring protocol formats is critical for many security applications. However, existing format-inference techniques often miss many formats, because almost all of them are in a fashion of dynamic analysis and rely on a limited number of network packets to drive their analysis. If a feature is not present in the input packets, the feature will be missed in the resulting formats. We develop a novel static program analysis for format inference. It is well-known that static analysis does not rely on any input packets and can achieve high coverage by scanning every piece of code. However, for efficiency and precision, we have to address two challenges, namely path explosion and disordered path constraints. To this end, our approach uses abstract interpretation to produce a novel data structure called the abstract format graph. It delimits precise but costly operations to only small regions, thus ensuring precision and efficiency at the same time. Our inferred formats are of high coverage and precisely specify both field boundaries and semantic constraints among packet fields. Our evaluation shows that we can infer formats for a protocol in one minute with >95% precision and recall, much better than four baseline techniques. Our inferred formats can substantially enhance existing protocol fuzzers, improving the coverage by 20% to 260% and discovering 53 zero-days with 47 assigned CVEs. We also provide case studies of adopting our inferred formats in other security applications including traffic auditing and intrusion detection.

1 INTRODUCTION

Network protocols define how computing systems are connected. Thus, security vulnerabilities in network protocols may have devastating consequences. For example, the WannaCry attack, which was caused by a protocol vulnerability, led to over $8 billion loss across 150 countries [3]. To aid automated security analysis for network protocols, a formal specification of packet formats is often mandatory — it enables security testing by facilitating the generation of legitimate network packets for security testing [45, 66]; they are the foundations for protocol model checking [27, 64] and formal verification [28]; and they can guide automated code generation with strong guarantees [72].

Anonymous Submission to ACM CCS 2022, Due 2 May 2022, Los Angeles, USA
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

However, while protocols may have their specification documents in natural languages, formal, or machine-readable, packet formats are often not available, and even when they are, they may be incomplete or inaccurate [25]. Therefore, automatically inferring formal protocol formats is of importance. There are three typical scenarios. First, the protocol implementation is not accessible but network packets are available. In this case, network trace analysis [36, 50, 51, 61, 74, 75, 80] are proposed. They leverage statistical analysis and machine learning to infer how a packet can be divided into fields. Since the underlying techniques have inherent uncertainty, the quality of inferred formats tends to be insufficient to drive many security applications such as protocol fuzzing. We call them the *category-one* techniques.

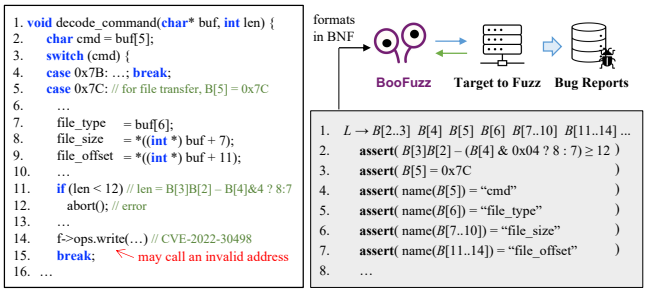
In the second scenario, the executable code of a protocol and a set of valid packets are available. Dynamic program analysis [29, 30, 38, 46, 47, 58–60, 77] then trace how individual packet bytes are propagated when running the code on the provided packets. The protocol formats then can be inferred from the data/control flow relations collected at runtime. For example, a typical rule to infer a raw data field is that consecutive bytes in the field are accessed by the same instruction [29, 38, 58]. These techniques can precisely infer the syntax of provided packets and denote the state-of-the-art. In some cases (e.g., [38]), semantics constraints, for example, those describing correlations across packet fields like the sum of fields A and B cannot exceed a certain threshold, can be inferred as well. However, the inferred formats are often incomplete when the provided packets do not have good coverage of all possible formats. We call them the *category-two* techniques.

We focus on the third scenario, in which the source code of a network protocol is available. As we will show in Section 6, open-source protocol implementations have many zero-day vulnerabilities. Without precise formal formats, existing protocol fuzzers such as BooFuzz [14] can hardly find them. In a recent work FRAME-SHIFTER [48], critical bugs were found by fuzzing HTTP/1 and HTTP/2, whose implementations are publicly available. While the authors manually crafted the protocol formats, automatic format inference can generalize their method to other protocols. In network traffic auditing and attack detection, e.g., using Wireshark [18] and Snort [13], substantial manual efforts are still needed to write dedicated protocol parsers for Wireshark and Snort even when a protocol is open-sourced. In contrast, with our inferred formal formats, Wireshark and Snort can be easily extended.

In this paper, we focus on the third scenario and develop a static program analysis to produce formal protocol formats, including both syntax and semantics, from the source code. We call it a protocol lifting technique, belonging to *category-three*. We resort to static analysis in order to address the coverage problem in dynamic analysis. Meanwhile, high accuracy can be achieved as it adopts a path-sensitive analysis. We produce BNF-like protocol formats. While BNF [35] is a common language to describe syntax, we enhance it to include first-order-logic semantic constraints across protocol fields. As we will show in §4, lifting source code to protocol formats is highly challenging. First, the traditional data-flow analysis that aggregates analysis results of multiple program paths at their joint point yields very poor results, whereas path-sensitive analysis that considers individual paths separately is prohibitively expensive due to path explosion. Second, the inferred formats are mostly out of order for human interpretation, which is highly undesirable as humans are important consumers of the formats in security applications.

To address the challenges, we develop a novel static analysis. In particular, we develop abstract interpretation rules that can derive an abstract format graph (AFG) from the source code. AFG can be considered as a transformed control flow graph. It precludes statements that are irrelevant to packet formats. It further merges program subpaths that are irrelevant to formats so that path-sensitive analysis is not performed on the merged places. Meanwhile, it retains sufficient information such that a localized but precise path-sensitive analysis can be performed on the unmerged parts of the graph. Therefore, it mitigates the path-explosion problem without losing analysis accuracy. The AFG is further unfolded and reordered to generate BNF-style production rules and first-order-logic formulas that describe semantic constraints across protocol fields. In summary, we make the following four contributions:

- We develop an abstract interpretation method that produces a novel representation, namely the abstract format graph, to facilitate format inference.
- We propose a localized graph unfolding algorithm that can perform precise path-sensitive analysis in small AFG regions to significantly mitigate path explosion.
- We devise a graph reordering algorithm that translates an unfolded AFG to the commonly-used BNF so that our inferred formats can be widely applied in practice.
- We implement our approach as a tool, namely Netlifter, to infer packet formats from protocol parsers written in C. We evaluate it on a number of protocols from different domains. Netlifter is highly efficient as it can infer formats in one minute. Netlifter is highly precise with a high recall as its inferred formats uncover $\geq$ 95% formats with $\leq$ 5% false ones. In contrast, the baselines, often miss >50% of formats and, sometimes, produce >50% false ones. We use the inferred formats to enhance grammar-based protocol fuzzers, which are improved by 20%-260% in terms of coverage and detect 53 zero-day vulnerabilities with 47 assigned CVEs. Without our formats, only 12 can be found. We also provide case studies of adopting our approach in traffic analysis and intrusion detection. Netlifter is publicly available [20].



(a) Code snippet (b) Workflow of protocol fuzzer

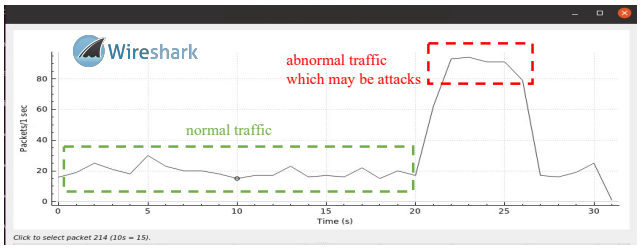
Figure 1: (a) Simplified code that parses the file-transfer command. (b) The typical workflow of protocol fuzzers with a snippet of the format inferred by Netlifter, in which the first row is a BNF production rule denoting syntax (e.g., field partitioning) and the remaining denote semantic constraints.

2 MOTIVATION

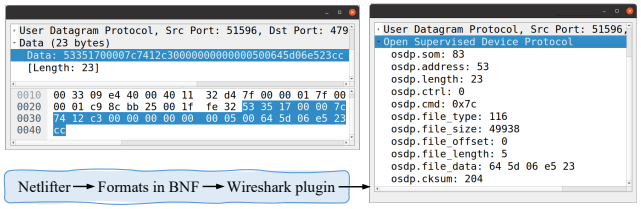
We use an open-source protocol, namely Open Supervised Device Protocol (OSDP), to illustrate the limitations of existing methods and how our technique can facilitate various security applications. OSDP is an access control communications standard developed by the Security Industry Association to improve interoperability among access control and security products. Although it is an open-source protocol, its full specification is not publicly available. The only available document [4] lacks many details. For instance, it includes the formats for only 7 out of the 27 supported commands.

The implementation of OSDP is vulnerable. Figure 1(a) shows a code snippet related to a zero-day bug found by a protocol fuzzer enhanced by our approach. The code shows part of the packet parsing function. The variable buf is a byte array representing the OSDP packet, and we use $B[i]$ to represent the $(i+1)$th byte in the packet. The bug is at Line 14. It may invoke an invalid function pointer *f->ops.write*, which could lead to a crash or be exploited for DoS or ROP attacks. There are multiple ways to avoid such attacks. The first one is to use fuzzing techniques to find bugs in its implementation and have them fixed before exploitation. The second is to provide OSDP support in network traffic analysis and attack detection tools such that the attack can be analyzed and further prevented. However, existing methods fall short as discussed below.

Standard Network Fuzzing Can Hardly Find the Zero-day. Different from stand-alone application fuzzers such as AFL [19], network fuzzers, such as BooFuzz [14], often operate in a client-server architecture. The server runs the target protocol implementation. The client leverages grammar-based fuzzing to generate packets as per the formats, send the packets to the server, receive responses, and generate new packets to fuzz the target. However, the effectiveness of these fuzzers hinges on the protocol formats. When the formats are not available like in our OSDP case, they quickly degenerate into traditional greybox fuzzers that arbitrarily mutate bits or bytes. Such mutated packets can hardly pass many input validity checks in the code. For example, in Figure 1(a), to expose the bug, a fuzzer has to get through the check at Line 11, which is a complex relation across multiple fields as shown in the comment. As a result, standard network fuzzers fail to find the bug when an imprecise or incomplete format of OSDP is provided.



(a) Abnormal traffic observed by Wireshark



(b) Packet details using original Wireshark (c) Details using our enhanced one

Figure 2: Network traffic auditing via Wireshark.

Lack of Support for OSDP in Wireshark and Snort. We can also rely on network traffic analyzers, e.g., Wireshark [18], and attack detection tools, e.g., Snort [13], to ensure security. However, both Wireshark and Snort do not support OSDP. Assume Wireshark is deployed at the gateway. It detects abnormal traffic as highlighted in the red box in Figure 2(a). Note that in the diagram the x-axis is time and the y-axis denotes the amount of traffic per second. However, the traffic is not interpretable for Wireshark as OSDP is not supported. Instead, the OSDP packets are treated as raw data bytes as shown in Figure 2(b). Thus, it is hard to analyze the packet details and determine which device launches the attack.

2.1 Limitations of Existing Techniques

A way to address the aforestated defense insufficiency is to infer the protocol formats. As discussed in §1, existing techniques fall into categories one and two. Category-one infers formats from a set of network packets. For example, a recent method NetPlier [80] leverages a probabilistic analysis on network packets to determine a keyword field, i.e., the field identifying the packet type, by computing the probabilities of each byte offset. Once the keyword field is determined, it clusters packets according to the value of the field and applies multi-sequence alignment to derive message format. However, real-world network packets suffer from all sorts of distribution biases, e.g., lacking some kinds of messages due to their rare uses in practice, leading to sub-optimal results.

For instance, NetPlier partitions the first four bytes of an OSDP packet as | 0x53 0xff | 0x29 | 0x00 | ... |, which mistakenly places the first two bytes into the same field and splits $B[2]$ and $B[3]$ into two different fields while $B[2..3]$ (with the value of $B[3]B[2]$ that represents a two-byte integer with $B[3]$ the most significant byte and $B[2]$ the least.) should be a single field representing the packet length. However, since most input packets are shorter than 255, $B[3]$ is always zero while $B[2]$ has different values in different packets. Thus, these two bytes follow different distributions in the packet samples, and NetPlier incorrectly regards them as separate fields. Moreover, NetPlier does not infer semantic constraints such as the condition at Line 11 in Figure 1(a). Such imprecise formats prevent a grammar-based protocol fuzzer from finding the zero-day (see §6) and fail to enhance Wireshark and Snort (see Appendix B).

Category-two methods dynamically analyze protocol execution using a set of input packets. AutoFormat [58] is a representative. It leverages the observation that most packet parsers utilize top-down parsing such that they invoke a function to parse a sub-structure. Therefore, the dynamic call graph in parsing a packet discloses its structure. However, the function call hierarchy may not be sufficiently fine-grained to disclose detailed packet formats. Similar to NetPlier, it does not infer semantic constraints across fields, such as the one at Line 11 in Figure 1(a). As dynamic analysis, the inferred format may be incomplete, depending on the coverage of the input packets that drive the dynamic analysis. For instance, in our evaluation, Autoformat misses 15 out of the 27 packet types because these types of packets do not appear in regular workloads.

Some category-two techniques, e.g., Tupni [38], can precisely infer semantic constraints among packet fields. However, as dynamic analyses, they suffer from the innate coverage problem. As per our results, the inference results of Tupni may miss >50% of possible formats. The problem is that if the program executions analyzed by Tupni do not cover the file-transfer command, i.e., Lines 5-15 in Figure 1(a), Tupni will not generate formats for the command. Without the formats, it is hard for a fuzzer to generate packets that can pass the validity check at Line 11 and expose the bug at Line 14.

2.2 Our Solution and Security Applications

Observing that the source code discloses substantial information about packet formats, we propose a category-three method that lifts the source code of OSDP to the protocol formats. For instance, Line 5 of the code in Figure 1(a) indicates that the command code for file transfer is 0x7c. Line 7 indicates that $B[6]$ is a field representing the file type to transfer. Lines 8-9 load two four-byte integers to variables file_size and file_offset and thus indicate that there are two four-byte fields, one from $B[7]$ to $B[10]$ and the other from $B[11]$ to $B[14]$, meaning the size and the offset of the file to transfer, respectively. In addition to the syntactic information (e.g., field partitioning), the code also discloses the semantic relations across fields. For instance, the if-statement at Line 11 implies a cross-field constraint dictating that if $B[4]$ & $4 = 0$, a valid packet must satisfy the constraint $B[3]B[2] - 7 \geq 12$ or, otherwise, $B[3]B[2] - 8 \geq 12$.

We extract the above syntactic and semantic information via static analysis and produce a BNF-style production rule in Figure 1(b). Our lifted formats are both precise and of high coverage. In terms of precision, we precisely identify each field and its name as shown in Figure 1(b) and, meanwhile, also specify the field constraints as first-order-logic formulas. In terms of coverage, since we do not rely on any input packets like category-one and category-two techniques, any format included in the source code will be inferred. We can use the lifted formats to support many applications.

Application 1: Finding Zero-days by Network Fuzzing. We leverage a theorem prover such as Z3 [40] to produce valuations for individual packet fields, such as the $B[i]$'s in Figure 1(b), which satisfy the semantic constraints. The generated packets can pass the check at Line 11 in Figure 1(a), thereby enabling the discovery of the CVE at Line 14. In addition, our inferred packet format is of high coverage and allows the fuzzer to generate diverse packets to

improve test coverage. In particular, the vulnerable code can only be reached when the packet is a file-transfer command, i.e., the 0x7c branch of the switch statement (Line 5). If the format is not covered, the chance that a fuzzer can mutate a packet of different types to a valid file-transfer command is very slim. Our evaluation shows that the lifted formats can improve the coverage of fuzzing by 20-260% and allow us to detect 41 more zero-days, compared to using the inferred specifications by category-one and category-two methods, which can only detect 12 zero-days. Note that we do not claim direct contributions to fuzzing. Instead, our approach is orthogonal to existing fuzzing methods that rely on packet formats.

Application 2: Network Traffic Auditing. Wireshark is the foremost protocol analyzer to ensure network security for hundreds of protocols [18]. Supporting a new protocol in Wireshark can be achieved by providing an extension, which is usually a library to parse protocol packets. We develop an extension generator that takes a lifted format as input and generates the corresponding Wireshark extension. Figure 2(c) shows that with the generated extension, Wireshark can look inside an OSDP packet sampled from the abnormal traffic. Our lifted format provides not only precise packet syntax but also informative field names extracted from variable names. With Wireshark, we observe that all packets during the abnormal traffic have the field osdp.address=35 and osdp.cmd=0x7c, indicating they are all from a device with the id #35 via the file-transfer commands. As will be shown in our evaluation, category-two approaches miss over 50% of possible fields. Extensions built from these incomplete formats would render Wireshark failing to process many received packets. In addition, they can hardly provide field names that are as informative as the ones we can provide.

Application 3: Network Intrusion Detection. Due to the space limit, we put discussions of this application in Appendix B.

Remark. While our inferred formats have high precision and recall close to 100%, like all previous works, there may still be missing or wrong formats, due to the inherent limitations of static program analysis (see §9). However, the inferred format still matters in practice, because many downstream applications do not require perfect formats. For example, although format inaccuracies cause degraded efficacy improvement in fuzzing, the performance may still be far better than without any formats or having low-quality formats. This is similar for network traffic auditing and intrusion detection.

3 BACKGROUND AND OVERVIEW

This section provides some background knowledge of our approach and overviews the lifting procedure, in order to facilitate the later discussion with more complexity.

Protocol Format vs. Protocol Specification. Generally, the specification of a protocol consists of protocol formats and protocol state machines [65]. Protocol formats are often specified using a grammar in BNF, which specifies how a network packet, i.e., a bit or byte stream, can be dissected into multiple segments, i.e., fields, and specifies the semantic constraints the fields need to satisfy. For example, Figure 3(b) shows a typical BNF-style format of OSDP packets. The productions specify how an OSDP packet can be divided into multiple fields such as *som*, *address*, and so on. Like many previous works [29, 30, 36, 38, 46, 47, 50, 51, 58–61, 74, 75, 77, 80], Netlifter focuses on inferring the protocol formats.

Protocol state machines, on the other hand, specify the state transitions of a network server upon receiving or sending a specific network packet. For instance, a TCP server may transition from the state SYN-SENT, meaning it waits for a matching connection, to the state ESTABLISHED, meaning a connection has been established, upon receiving an acknowledgment packet. There have been many works also focusing on state machine inference [23, 32, 33, 37, 41, 53, 55, 62, 69, 76, 80, 81]. Typically, these works accept a set of network packets as input and produce the protocol state machines. While Netlifter only focuses on the formats, since the formats are sufficient to produce a number of packets, one can feed the packets into the aforementioned techniques for state machine inference.

Static Program Analysis. Static analysis analyzes program behaviors without executing programs and often has various forms such as dataflow analysis and abstract interpretation. As pointed out by Cousot and Cousot [34], while dataflow analysis and abstract interpretation are in different forms, they are equivalent when used to compute sound results. Basically, this is because both of them use abstract values to approximate program behavior. A complete/sound analysis uses abstract values, which are often formulas over predefined symbols, to under-/over-approximate concrete values that may assign to program variables at runtime. For instance, in our work, the abstract value of a variable is a formula over bytes, e.g., $B[0], B[1], \ldots,$ in a network packet. Here, $B[i]$ is a byte ranging from 0x00 to 0xFF and, thus, over-approximates all possible values of the $(i+1)$th byte.

A static analysis is often defined by a set of transfer functions and merging functions over abstract values. A transfer function specifies how we compute an abstract value when visiting a program statement. For instance, given the abstract values of two variables, e.g., $x = B[1], y = B[2]$, the transfer function of the statement $z = x + y$, accepts the abstract values of x and y as the input and outputs the abstract value of z, which typically is $B[1] + B[2]$.

When a variable is assigned multiple abstract values in two program paths, at the joint point of the two paths, we use a merging function to compute a merged abstract value. For instance, assume that we compute $x = B[1]$ and $x = B[2]$ in two different paths and Θ is an operator returns either of its operands. At the joint point, the merged abstract value of x could be $\Theta(B[1], B[2])$. This value is sound as it over-approximates the value of x, saying that x is either $B[1]$ or $B[2]$. However, it is not complete or not precise, because it loses the path information, i.e., from which path $x = B[1]$ (or $B[2]$).

A static analysis can be performed with varying degrees of precision. In this work, we choose to perform a path-sensitive static analysis, which is of high precision as it can distinguish abstract values from different paths. To this end, one often needs to enumerate all paths in a program, just like symbolic execution [49], which, however, suffers from the notorious path-explosion problem due to the exponential number of paths in a program. In this work, we aim to significantly mitigate this problem by introducing a special merging operator, Θ_K, as explained later.

Input & Output of Netlifter. The input of our static analyzer is the source code of a *top-down protocol parser* [21] written in C. A top-down parser applies each production rule in a BNF-style format to incoming bytes of the network packet, working from the left-most symbol of a production rule and then proceeding to the next

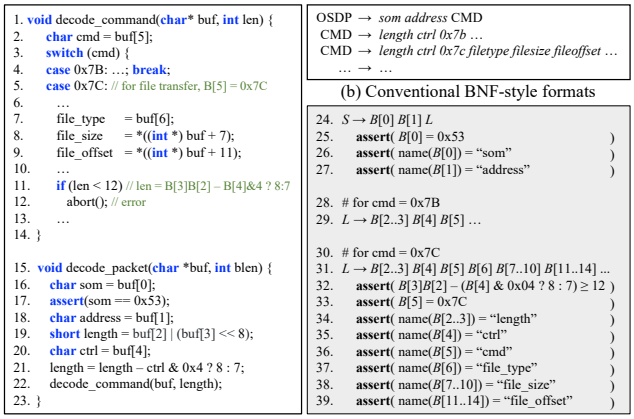


Figure 3: Extended example for OSDP.

production rule for each non-terminal symbol encountered [22]. Given the parsing function of a protocol, e.g., *parse(char* buf, int len) { ... }*, the user annotates the parameters, i.e., the buffer variable, *buf*, that contains the network packet to parse, and the integer variable, *len*, which stands for the packet length. Except for the two annotations, Netlifter is fully automated.

The output of Netlifter is the protocol format defined below. The format is similar to common BNF so that it aligns well with existing standards in formally describing protocol formats.

Definition 1 (Protocol Format). The format includes syntax and semantics. The syntax is denoted by production rules in BNF, where each rule is a sequence of consecutive bytes. Semantics is described by non-recursive first-order-logic (FOL) constraints with two special functions, name(...) and repeat(...), which are explained in the example below. The format satisfies three properties:

(1) Each terminal symbol in the grammar is either $B[i]$ or $B[i..j]$, which is a bit-vector standing for the $(i+1)$th byte or a range of bytes from $B[i]$ to $B[j]$;
(2) Each production rule is associated with a set of assertions that assert FOL constraints over the terminals in this rule. The constraints must not conflict with each other.
(3) Each assertion contains only a single atomic constraint that does not contain any connectives $\wedge$ or $\vee$.

Example (Input of Netlifter). Figure 3 extends the example in Figure 1. It shows a simplified OSDP parser starting from Line 15. The packet is a byte array stored in *buf* and the array length is *blen*. The user needs to annotate the two variables. The parser with the annotations is the input of Netlifter. In Lines 16-21, the parser loads the first five bytes into the variables *som*, *address*, *len*, and *ctrl*, where *som* stands for "start of message" and is used to identify OSDP packets. The remaining code invokes the function *decode_command* to parse an OSDP command as explained in Figure 1. □

Example (Output of Netlifter). Figure 3(b) shows a typical BNF-style format of OSDP, which is often manually constructed. The output format of Netlifter is shown in Figure 3(c), which closely resembles the manually constructed BNF in (b). The first rule in (c) resembles the first rule in (b), where we correctly determine that the first two bytes, $B[0]$ and $B[1]$, are two separate fields, corresponding to the fields *som* and *address* in (b). Similarly, the second and third rules in (c) resemble the two CMD rules in (b), where, besides single-byte fields, we also correctly determine multi-byte fields including $B[2..3]$, $B[7..10]$, and $B[11..14]$, corresponding to the fields *length*, *filesize*, and *fileoffset* in (b).

The output format also associates each rule with two kinds of assertions. One kind, such as Line 25 and Lines 32-33, specifies the semantic constraints among packet fields. They are inferred from branching conditions in the code. When we infer a constraint including a value like $B[3]B[2]$, it indicates a two-byte field with $B[3]$ the most significant byte and $B[2]$ the least. In other words, in addition to *field boundaries*, our format also expresses the *endianness*, whereas the standard BNF cannot. Netlifter also describes semantic constraints not expressible in standard BNF such as the one in Line 32. All constraints have the *bit-level precision*. For instance, the expression $B[4]$ & 4 in Line 32 computes the third bit of $B[4]$.

The other kind, such as Lines 26-27 and Lines 34-39, specifies the field names, which provide *high-level field semantics* for us to understand the format. In addition to those in the example, we also produce many other names such as name($B[i..j]$) = 'timestamp'/'checksum' to indicate a timestamp/checksum field. As explained later, we infer such high-level semantics using the names of program variables or library APIs. □

In addition to the example above, we elaborate on several places where our format is more expressive than the standard BNF.

Direction and Variable-Length Fields. A direction field locates another field and is often a length field, whose value encodes the variable length of a target field [30]. For instance, we may produce a rule $S \rightarrow B[0]B[1..B[0]]$, where $B[1..B[0]]$ is a variable-length field whose length is determined by the direction field $B[0]$.

Repetitive Fields. Our format can also specify repetitive fields and how many times a field repeats. For instance, we may produce the production rule $S \rightarrow B[0]B[1..2]B[3]$ with three assertions: (1) **assert**$(B[0] = B[3] = 0x00)$, (2) **assert**$(B[1]B[2] \neq 0x0000)$, and (3) **assert**$(\text{repeat}(B[1..2]) = 3)$. The first assertion constrains the first and last byte. The second constrains the field $B[1..2]$ in middle. The third states that the middle field repeats three times. When generating packets based on the rule, we first generate a packet satisfying the first two assertions, e.g., $0x00$ $0x0001$ $0x00$. Due to the third assertion, we insert another two fields satisfying the same constraints as $B[1..2]$, e.g., $0x00$ $0x0001$ $0x0011$ $0x0101$ $0x00$.

4 TECHNICAL CHALLENGES

This section discusses two prominent challenges as well as our ideas for addressing them. It provides a context for the detailed discussion later and is driven by the crafted running example in Figure 4. Figure 4(a) shows a protocol parser and Figure 4(e) shows an ideal lifted format. The code implies complex cross-field constraints which are clearly represented in Figure 4(e). We can see that a packet of the protocol contains three segments L_1, $L_2|L_3$, and $L_4|L_5$. The segment L_1 has two fields, a *code* field (reflected by Lines 10-11 in the code) and a *state* field (reflected by Lines 14-16 in the code). Both L_2 and L_3 consist of a single field located at byte offset 3, and differ only in the field value (reflected by Line 13 in the code). Both L_4 and L_5 consist of three single-byte fields and differ in the semantic constraints (reflected by Lines 3-8 in the code).

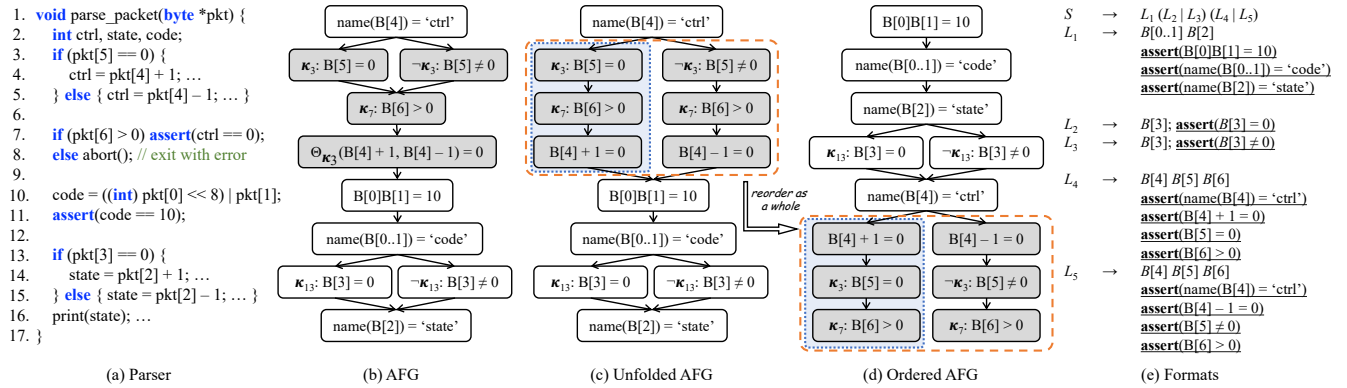


Figure 4: A crafted running example.

Challenge 1: Insufficiency of Traditional Static Analysis. Traditional static analysis is path-insensitive and merges analysis results from different paths at their joint point to achieve scalability. As introduced before, such merging yields over-approximation and incurs low precision. For example, the abstract values of *ctrl* from the two branches at Lines 4 and 5, respectively, are merged at Line 6, yielding $ctrl = B[4] + 1 \vee ctrl = B[4] - 1$. As such, we lose the correlation between $B[4]$ and $B[5]$ as the precise value of *ctrl* should depend on the value of $B[5]$ due to the if-statement at Line 3. In consequence, the resulting format will lose the correlations between $B[4]$ and $B[5]$, while in the ideal format shown in Figure 4(e), the production rules L_4 and L_5 include such correlation, i.e., $B[4] + 1 = 0 \Leftrightarrow B[5] = 0$ and $B[4] - 1 = 0 \Leftrightarrow B[5] \neq 0$.

A typical solution is to use a path-sensitive static analysis that separately analyzes individual paths and does not merge results from multiple branches. Lifting is thus reduced to enumerating paths, each constituting a production rule. In our example, there are four paths that denote valid packets, i.e., (P1) … → 4 → … → 14 → …, (P2) … → 5 → … → 14 → …, (P3) … → 4 → … → 15 → …, and (P4) … → 5 → … → 15 → …. Thus, the lifted format has four rules, each of which corresponds to a path constraint. For example, the format for the path P1 is shown below.

$$S \rightarrow B[0..1]\ B[2]\ B[3]\ B[4]\ B[5]\ B[6]$$
$$\textbf{assert}(B[5] = 0); \textbf{assert}(B[6] > 0); \textbf{assert}(B[4] + 1 = 0);$$
$$\textbf{assert}(B[0]B[1] = 10); \textbf{assert}(B[3] = 0);$$

Enumerating program paths incurs the notorious path-explosion problem, which has two consequences: (1) the analysis is not scalable and (2) the lifted format has an explosive number of semantic constraints. For example, due to path explosion, KLEE [31], a state-of-the-art path-sensitive static analyzer, cannot finish analyzing Linux's implementation of L2CAP, a Bluetooth protocol containing a few thousand lines of code, within twelve hours.

Solution 1: Localized Path-Sensitive Analysis. We observe that in lifting, path-sensitivity is only needed in certain places. In our example, we only want to analyze sub-paths 3 → 4 → 7 and 3 → 5 → 7 separately such that we can generate production rules L_4 and L_5 in Figure 4(e). *The criterion to determine a local code region for path-sensitive analysis is that the path conditions within the region and their branches have inter-dependencies.* For example, the condition at Line 3 determines the value of *ctrl*, which is checked inside the true branch at Line 7, allowing Lines 3-8 to form a region for path-sensitive analysis. In contrast, Lines 10-16 have no dependencies on Lines 3-8 and, thus, are considered separately. In Section 5, we will discuss how we use a new selection operator and a novel representation called abstract format graph (AFG) to identify the regions for localized path-sensitive analysis.

Challenge 2: Handling Out-of-Order Fields. Protocol parsers may not parse network packets in strict byte order. Hence, if a naive lifting algorithm directly derives format from code, for example, generating production rules following the order that the bytes are accessed along program paths, the resulting format may have out-of-order fields, which do not comply with the BNF standard. For example in Figure 4(a), the parser parses bytes $B[4..6]$ before $B[0..3]$. Therefore, we have to break the program order. This requires us to reorder the bytes such that they follow the byte order while not violating program semantics. For example, in Lines 3-8 in Figure 4 (a), the access of *pkt[4]* occurs after that of *pkt[5]*. One cannot simply relocate Line 4 and the else branch in Line 5 to in front of Line 3, because the resulting program is broken as shown in the following.

```
ctrl=pkt[4]-1;  ctrl=pkt[4]-1;  if (pkt[5]==0)  ...
```

Solution 2: Graph-Based Reordering. We propose to first abstract the code to the aforementioned AFG that models only the packet format-related behaviors and precludes the rest. As such, we do not need to transform the program which is complex and unnecessary. An algorithm is developed to ensure dependencies can be respected during reordering.

5 DESIGN

Figure 4(a-e) presents the workflow of Netlifter. Given the code of a protocol, abstract interpretation is performed to construct an abstract format graph (AFG). Path-sensitive analysis is performed in selected local regions of AFG to produce an unfolded AFG, which is further reordered and post-processed to produce the lifted formats.

5.1 Abstract Format Graph

AFG is a directed acyclic graph representing first-order-logic constraints. The AFG of a constraint ρ is inductively defined by $\text{AFG}(\rho)$:

(1) $\text{AFG}(\textit{atomic constraint}) = \text{Vertex}(\textit{atomic constraint})$;
(2) $\text{AFG}(\rho_1 \wedge \rho_2) = \text{AFG}(\rho_1) \bowtie \text{AFG}(\rho_2)$;
(3) $\text{AFG}(\rho_1 \vee \rho_2) = \text{AFG}(\rho_1) \uplus \text{AFG}(\rho_2)$.

Generally, a vertex of AFG is an atomic constraint that does not contain any connectives $\wedge$ or $\vee$, and an edge means logical conjunction. In the definition, the first rule returns a single vertex for any atomic constraint. The second creates a graph for conjunction by connecting all *exit vertices* (vertices without outgoing edges) of $\text{AFG}(\rho_1)$ to all *entry vertices* (vertices without incoming edges) of $\text{AFG}(\rho_2)$. The third creates a graph for disjunction by simply creating a union of the two graphs, which contains the vertices and edges from both. The following lemma states the equivalence relation between the graph $\text{AFG}(\rho)$ and the constraint ρ. In other words, $\text{AFG}(\rho)$ is an equivalent graphic representation of the constraint ρ. We put the proofs of all our lemmas in Appendix C.

Lemma 5.1. *Given $AFG(\rho)$ with n paths, we have $\rho \equiv \bigvee_{i=1}^{n} \rho_i$ where each ρ_i equals the conjunction of all constraints in an AFG path.*

Example. Consider the constraint $\rho \equiv (a \vee b) \wedge c \wedge (d \vee e)$. By definition, $\text{AFG}(\rho)$ is a directed graph with five nodes, which respectively correspond to the five atomic constraints a, b, c, d, and e. The AFG also contains four edges respectively from a to c, from b to c, from c to d, and from c to e. The AFG has four paths, respectively representing four constraints, $\rho_1 = a \wedge c \wedge d$, $\rho_2 = a \wedge c \wedge e$, $\rho_3 = b \wedge c \wedge d$, and $\rho_4 = b \wedge c \wedge e$. Apparently, we have $\rho_1 \vee \rho_2 \vee \rho_3 \vee \rho_4 \equiv \rho$. Thus, we say the $\text{AFG}(\rho)$ is an equivalent graphic representation of the constraint ρ. □

5.2 Abstract Interpretation

The static analysis derives an AFG denoting path constraints related to the packet format. It features a new selection operator at the joint point of branches, which enables localized path-sensitive analysis.

Abstract Language. For clarity, we use a C-like language in Figure 5 to model our target programs. A program in the language has an entry function that parses an input network packet, *pkt*, which is a byte array. The parsing function often has a parameter specifying the packet length, *len*, to avoid out-of-bounds access during parsing. The language contains assignments, binary operations, statements that read bytes from the packet, assertions, branching, and sequencing. Each branching statement is labeled by a unique identifier κ. Although we do not include function calls or returns for discussion simplicity, our system is inter-procedural as a call statement is equivalent to a list of assignments from the actual parameters to the formals, and a return statement is an assignment from the return value to its receiver. The language includes statements reading bytes from the packet but does not include statements that store values into the packet. This is because, for parsing purposes, the input packet is often read-only. Note that the abstract language serves for demonstrating how we address the challenges discussed in §4. Thus, for simplicity, we abstract away some common program structures, e.g., pointers and loops, from the language. Dealing with these structures is not our technical contribution. In §5.5, we discuss how we handle them in our implementation.

Abstract Domain. An abstract value of a variable represents all possible concrete values that may be assigned to the variable during program execution. The abstract domain specifies the limited forms of an abstract value. In our analysis, the abstract value of a variable v is denoted as $\tilde{v}$ and defined in Figure 6. An abstract value could be a constant or a special value *length* that represents the packet length. The $(\tilde{v}+1)$th byte of the input packet is $B[\tilde{v}]$. We introduce a new selection operator Θ_κ such that $v = \Theta_\kappa(v_1, v_2)$, which means that when the if-statement at κ takes the true branch, we have $v = v_1$, $v = v_2$ otherwise. One may find that the operator Θ_κ is similar to the operator ϕ in the classic SSA code form [39] because both of them merge values from multiple branches. We note that Θ_κ differs from ϕ in two aspects. First, in the SSA form, $v = \phi(v_1, v_2)$ is always placed at the end of a branching statement, whereas in our analysis $v = \Theta_{\kappa_i}(v_1, v_2)$ represents an abstract value of the variable v and is propagated to many other places where the variable v is referenced. Second, since $v = \Theta_\kappa(v_1, v_2)$ may be used at any place in the code, we use the subscript κ to record the branching statement where it originates. This is a critical design for the next step, i.e., the localized graph unfolding, as illustrated later. An abstract value can also be a first-order logic formula over other abstract values. To ease the explanation, we only support binary formulas.

Figure 6 lists the rules that normalize expressions over abstract values. Rule (1) states that we do not need a Θ_κ operator if we merge two equivalent values. Rules (2-3) state that any operation with a Θ_κ-merged value is equivalent to operating on each value merged by the Θ_κ operator. Rules (4-5) simplify nested Θ_κ operators.

Abstract Semantics. The abstract semantics describe how we analyze a given protocol parser. They are described as transfer functions of program statements. Each transfer function updates the program's abstract state, which is a pair $(\mathbb{E}, \mathbb{G})$. Given the set V of program variables and the set $\tilde{V}$ of abstract values, $\mathbb{E} : V \mapsto \tilde{V}$ maps a variable to its abstract value. We use $\mathbb{E}[v \mapsto \tilde{v}]$ to denote updating the abstract value of the variable v to $\tilde{v}$. $\mathbb{G}$ is the output AFG. Since AFG is an equivalent form of path constraint, we directly create AFG without computing the path constraint first.

Figure 7 lists the transfer functions as inference rules. In each rule, the part above the horizontal line includes a set of assumptions and, under these assumptions, the bottom part describes the abstract states before and after a statement S, in the form of $\mathbb{E}, \mathbb{G} \vdash S : \mathbb{E}', \mathbb{G}'$. Initially, we assign the special abstract value *length* to the variable

$$
\begin{array}{lcll}
\textit{Function } F & := & \textit{parse}(\textit{pkt}, \textit{len})\,\{\ S;\ \} & \\
\textit{Statement } S & := & v_1 \leftarrow v_2 & \textbf{::assign} \\
 & & \mid v_1 \leftarrow v_2 \oplus v_3 & \textbf{::binary} \\
 & & \mid v_1 \leftarrow \textit{pkt}[v_2] & \textbf{::read} \\
 & & \mid \textbf{assert}(v_1) & \textbf{::assertion} \\
 & & \mid \textbf{if}_\kappa\ (v)\ \{S_1;\}\ \textbf{else}\ \{S_2;\} & \textbf{::branching} \\
 & & \mid S_1; S_2 & \textbf{::sequencing}
\end{array}
$$

$$\oplus \in \{\wedge, \vee, +, -, >, <, =, \neq, \dots\}$$

Figure 5: Language of target programs.

$$
\begin{array}{lcll}
\textit{Abstract Value } \tilde{v} & := & c & \textbf{::constant} \\
 & & \mid \textit{length} & \textbf{::packet length} \\
 & & \mid B[\tilde{v}] & \textbf{::byte in packet} \\
 & & \mid \Theta_\kappa(\tilde{v}_1, \tilde{v}_2) & \textbf{::selection} \\
 & & \mid \tilde{v}_1 \oplus \tilde{v}_2 & \textbf{::binary operation}
\end{array}
$$

$$\frac{\tilde{v}_1 = \Theta_\kappa(\tilde{v}_2, \tilde{v}_2)}{\tilde{v}_1 = \tilde{v}_2}\ (1) \qquad \frac{\tilde{v}_1 = \Theta_\kappa(\tilde{v}_2, \tilde{v}_3)}{\tilde{v}_1 \oplus \tilde{v}_4 = \Theta_\kappa(\tilde{v}_2 \oplus \tilde{v}_4, \tilde{v}_3 \oplus \tilde{v}_4)}\ (2)$$

$$\frac{\tilde{v}_1 = \Theta_\kappa(\tilde{v}_2, \tilde{v}_3)}{B[\tilde{v}_1] = \Theta_\kappa(B[\tilde{v}_2], B[\tilde{v}_3])}\ (3)$$

$$\frac{\tilde{v}_1 = \Theta_\kappa(\Theta_\kappa(\tilde{v}_2, \tilde{v}_3), \tilde{v}_4)}{\tilde{v}_1 = \Theta_\kappa(\tilde{v}_2, \tilde{v}_4)}\ (4) \qquad \frac{\tilde{v}_1 = \Theta_\kappa(\tilde{v}_2, \Theta_\kappa(\tilde{v}_3, \tilde{v}_4))}{\tilde{v}_1 = \Theta_\kappa(\tilde{v}_2, \tilde{v}_4)}\ (5)$$

Figure 6: Abstract values.

$$\frac{\mathbb{E}=\emptyset \quad \mathbb{G}=\emptyset}{\mathbb{E},\mathbb{G}\vdash \mathit{parse}(\mathit{pkt},\mathit{len}):\mathbb{E}[\mathit{len}\mapsto \mathit{length}],\mathbb{G}}\ \textbf{init} \qquad \frac{\mathbb{E}(v_2)=\tilde{v}_2}{\mathbb{E},\mathbb{G}\vdash v_1\leftarrow v_2:\mathbb{E}[v_1\mapsto \tilde{v}_2],\mathbb{G}}\ \textbf{assign} \qquad \frac{\mathbb{E}(v_2)=\tilde{v}_2 \quad \mathbb{E}(v_3)=\tilde{v}_3}{\mathbb{E},\mathbb{G}\vdash v_1\leftarrow v_2\oplus v_3:\mathbb{E}[v_1\mapsto \tilde{v}_2\oplus\tilde{v}_3],\mathbb{G}}\ \textbf{binary}$$

$$\frac{\mathbb{E}(v_2)=\tilde{v}_2}{\mathbb{E},\mathbb{G}\vdash v_1\leftarrow \mathit{pkt}[v_2]:\mathbb{E}[v_1\mapsto B[\tilde{v}_2]],\mathbb{G}}\ \textbf{read} \qquad \frac{\mathbb{E}(v_1)=\tilde{v}_1}{\mathbb{E},\mathbb{G}\vdash \textbf{assert}(v_1):\mathbb{E},\mathbb{G}\bowtie \mathsf{AFG}(\tilde{v}_1)}\ \textbf{assertion} \qquad \frac{\mathbb{E}_1,\mathbb{G}_1\vdash S_1:\mathbb{E}_2,\mathbb{G}_2 \quad \mathbb{E}_2,\mathbb{G}_2\vdash S_2:\mathbb{E}_3,\mathbb{G}_3}{\mathbb{E}_1,\mathbb{G}_1\vdash S_1;S_2:\mathbb{E}_3,\mathbb{G}_3}\ \textbf{sequencing}$$

$$\frac{\mathbb{E}(v)=\tilde{v} \quad \mathbb{G}_1=\mathsf{AFG}(\tilde{v}) \quad \mathbb{G}_2=\mathsf{AFG}(\neg\tilde{v}) \quad \mathbb{E},\mathbb{G}\bowtie\mathbb{G}_1\vdash S_1:\mathbb{E}_\kappa,\mathbb{G}\bowtie\mathbb{G}_\kappa \quad \mathbb{E},\mathbb{G}\bowtie\mathbb{G}_2,\vdash S_2:\mathbb{E}_{\neg\kappa},\mathbb{G}\bowtie\mathbb{G}_{\neg\kappa}}{\mathbb{E},\mathbb{G}\vdash \textbf{if}_\kappa\ (v)\ \{S_1;\}\ \textbf{else}\ \{S_2;\}:\mathsf{mergeE}(\mathbb{E}_\kappa,\mathbb{E}_{\neg\kappa},\kappa),\mathbb{G}\bowtie(\mathbb{G}_\kappa\uplus\mathbb{G}_{\neg\kappa})}\ \textbf{branching}$$

procedure mergeE($\mathbb{E}_1, \mathbb{E}_2, \kappa$)
  foreach $(v, \tilde{v}_1) \in \mathbb{E}_1 \wedge (v, \tilde{v}_2) \in \mathbb{E}_2$ **do** $\mathbb{E}_1 \leftarrow \mathbb{E}_1[v \mapsto \Theta_\kappa(\tilde{v}_1, \tilde{v}_2)]$;
  return $\mathbb{E}_1$;

Figure 7: Inference rules and auxiliary procedure.

len, which represents the length of input network packet. The rules for assignment, binary operation, read operation, and assertion are straightforward. For instance, in the rule for assertions, the abstract value $\tilde{v}_1$ represents a constraint that must be satisfied. Therefore, we append the graph $\mathsf{AFG}(\tilde{v}_1)$ to the graph $\mathbb{G}$. This is equivalent to appending the constraint $\tilde{v}_1$ to the current path constraint.

The sequencing rule states that, for two consecutive statements, we analyze them in order, using the postcondition of the first statement as the precondition for the second. In the branching rule, $\mathbb{G}$ denotes the path constraint before the branching statement. $\mathbb{G}_1$ and $\mathbb{G}_2$ represent the branching condition and its negation. Thus, $\mathbb{G} \bowtie \mathbb{G}_1$ and $\mathbb{G} \bowtie \mathbb{G}_2$ represent the initial path constraints before the two branches. After analyzing the two branches, the resulting AFGs are assumed to be $\mathbb{G} \bowtie \mathbb{G}_\kappa$ and $\mathbb{G} \bowtie \mathbb{G}_{\neg\kappa}$. The branching rule states that, under these assumptions and after an $\textbf{if}_\kappa$-statement, we merge the abstract states from both branches. The procedure mergeE merges abstract values of the same variable via the Θ_κ operator. Graph merging is straightforward based on the definition of AFG, which is equivalent to merging path constraints of the two branches with the common prefix pulled out. Our merging is different from the value merging in traditional analyses due to the use of the selection operator. On one hand, merging allows achieving scalability as the number of values is no longer exponential of the number of statements. On the other hand, the selectors in abstract values can be unfolded to support path-sensitive analysis if needed.

Packet Fields. The abstract interpretation builds the AFG to represent the path constraints. As discussed in §3, from these constraints, it is direct to infer the endianness, field boundaries, and direction fields. For instance, if multiple consecutive bytes, e.g., $B[0]$ and $B[1]$ in Figure 4, belong to a single field, the field value, e.g., $B[0]B[1]$, will be computed and occur in the path constraint.

High-Level Field Semantics. We also extend our analysis to infer high-level field semantics, i.e., field names, using rich source code information. Such high-level semantics can help better understand a format, e.g., identifying checksum fields and distinguishing keywords and delimiters (both of which are constant fields). As illustrated in Figure 4, we can name a field (via some variable name) by adding extra path constraints. Formally, given the AFG $\mathbb{G}$ and a formula over a field $B[i..j]$, denoted as $\mathsf{f}(B[i..j])$, we name the field by $\mathbb{G} \bowtie \mathsf{AFG}(\mathsf{name}(B[i..j]) = \text{'var'})$ if there is a statement assigning $\mathsf{f}(B[i..j])$ to the variable var. In addition to variable names, we also leverage system APIs used in the code. For instance, if a field $B[i..j]$ is used in the system API, difftime(), it is likely to be a timestamp field. In our experience, this method helps us identify many special fields via names such as 'length', 'version', 'checksum', 'timestamp', etc. In our current implementation, we handle all standard C APIs. If there are multiple options for naming a field, we prefer the names inferred by system APIs because software developers may not be careful to name program variables. If there are still multiple options, we simply keep the first.

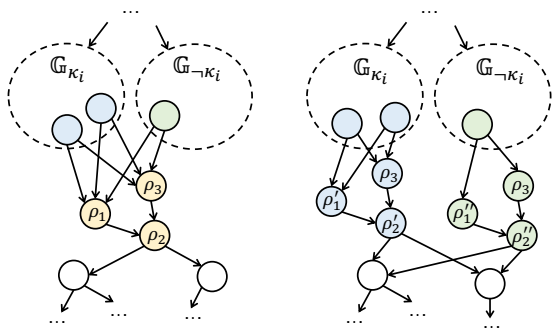
Example. Given the code in Figure 4(a), the abstract interpretation yields the AFG in (b) from top to bottom. After Line 5, we merge the two paths forked from Line 3 and get the path constraint: $\rho \equiv \mathsf{name}(B[4]) = \text{'ctrl'} \wedge (B[5] = 0 \vee B[5] \neq 0)$. By the branching rule, we do not compute the path constraint but directly create the equivalent AFG, i.e., the first two rows in Figure 4(b). We name the byte $B[4]$ 'ctrl' because the arithmetic results of $B[4]$ are assigned to the variable ctrl in both branches. The constraint $B[5] = 0 \vee B[5] \neq 0$ merges the branching constraints. Meanwhile, the abstract store is updated such that $\text{ctrl} = \Theta_{\kappa_3}(B[4] + 1, B[4] - 1)$. At Line 7, since the false branch aborts, we only consider the true branch, for which we add $B[6] > 0 \wedge \Theta_{\kappa_3}(B[4] + 1, B[4] - 1) = 0$ to the constraint ρ. This is equivalent to adding the third and fourth rows in Figure 4(b). At Lines 10-11, we add the constraint $B[0]B[1] = 10 \wedge \mathsf{name}(B[0..1]) = \text{'code'}$. This is equivalent to adding the fifth and sixth rows in Figure 4(b). We regard $B[0]$ and $B[1]$ as a single field as they are used in a single value $B[0]B[1]$.

Similarly, after Line 15, we merge the paths forked from Line 11 as the constraint $(B[3] = 0 \vee B[3] \neq 0) \wedge \mathsf{name}(B[2]) = \text{'state'}$ and append it to the path constraint ρ. This is equivalent to adding the last row in Figure 4(b). After Line 15, we update the value $\text{state} = \Theta_{\kappa_{13}}(B[2] + 1, B[2] - 1)$. In this example, the variable state is simply printed at Line 16 and never used in any if-statements or assertions. Hence, the merged value of state is abstracted away from the final constraint. Observe that the size of AFG is linear size with the number of statements. This is critical to scalability. □

Lemma 5.2. *Given a program in the language defined in Figure 5, the AFG produced by the abstract interpretation is sound and complete.*

5.3 Localized Graph Unfolding

Recall that path sensitivity is needed in localized regions during lifting (Challenge 1 in §4). Specifically, a code region that requires path sensitivity is identified as follows. *If a Θ_κ-merged value is later used in some path condition κ', the individual combinations of branch outcomes of κ and κ' need to be analyzed separately.* That is, path sensitivity is needed within the code regions of κ and κ'. On the other hand, many Θ_κ-merged values are not used in any later conditionals, the paths within the code region of κ do not need to be enumerated. That is, path sensitivity is not necessary.



(a) Before unfolding (b) After unfolding

Figure 8: Example of unfolding an AFG.

Specifically, given an AFG created by the abstract interpretation, we eliminate all Θ_κ-merged values by a localized graph unfolding algorithm shown in Algorithm 1. Assume that the AFG to unfold contains a list of Θ_κ operators, e.g., Θ_{κ_0}, Θ_{κ_1}, and Θ_{κ_2}. The algorithm eliminates Θ_{κ_i} one by one. For each Θ_{κ_i}, it works in two steps — slicing (Lines 3-7) and unfolding (Lines 8-11). To ease the explanation, we use Figure 8 for illustration. In Figure 8(a), without loss of generality, assume that we are unfolding Θ_{κ_i} in the AFG and that only the constraints ρ_1 and ρ_2 contain Θ_{κ_i}-merged values. The exiting vertices of $\mathbb{G}_{\kappa_i}$ and $\mathbb{G}_{\neg\kappa_i}$ are shown in the figure.

Slicing. This step delimits the next unfolding step to a local region in AFG. First, we find all exiting vertices of $\mathbb{G}_{\kappa_i}$ and $\mathbb{G}_{\neg\kappa_i}$. We then perform a forward graph traversal (e.g., depth-first search) from the exiting vertices. Denote the subgraph visited during the traversal as $\mathbb{G}_{\text{forward}}$. Second, we identify all vertices containing Θ_{κ_i}-merged values, e.g., ρ_1 and ρ_2 in Figure 8(a). A backward graph traversal from them yields a subgraph denoted as $\mathbb{G}_{\text{backward}}$. The overlapping part of $\mathbb{G}_{\text{forward}}$ and $\mathbb{G}_{\text{backward}}$, e.g., the yellow part in Figure 8(a), is the graph slice we will perform unfolding, denoted as $\mathbb{G}_{\text{slice}}$.

Unfolding. As illustrated in Figure 8(b), we copy the subgraph to unfold, obtaining $\mathbb{G}_{\text{slice}}$ and $\mathbb{G}'_{\text{slice}}$. The copy $\mathbb{G}_{\text{slice}}$ is connected to $\mathbb{G}_{\kappa_i}$, and by the definition of the merging operator, all the Θ_{κ_i}-merged values are replaced by its first operand. Similarly, the other copy $\mathbb{G}'_{\text{slice}}$ is connected to $\mathbb{G}_{\neg\kappa_i}$, and all the Θ_{κ_i}-merged values are replaced by its second operand. Since the subgraphs to unfold are limited in small local regions in practice, we significantly mitigate the path-explosion problem, which is sufficient to make our approach scalable. Note that we do not claim to have a theoretical bound on the size of subgraphs that need to be unfolded, as path explosion is still an open problem and cannot be completely addressed in theory, similar to all previous path-sensitive analyzers.

Lemma 5.3. *The unfolded AFG does not contain Θ_κ-merged values and represents an equivalent constraint as the original AFG.*

Example (continued). In Figure 4(b), the value merged by Θ_{κ_3} indicates that the branches forked at Line 3 need a path-sensitive analysis and delimits the analysis to the local region colored gray. To distinguish the two branches, the gray region in (b) is unfolded to two disjoint paths in (c), which eliminates the Θ-merged values and make the two semantic relations among $B[4]$, $B[5]$, and $B[6]$ explicit: $B[5] = 0 \wedge B[6] > 0 \Leftrightarrow B[4]+1 = 0$; and $B[5] \neq 0 \wedge B[6] > 0 \Leftrightarrow B[4] - 1 = 0$. In contrast, the $\Theta_{\kappa_{13}}$ value in variable *state* is never used in any conditional, suggesting that we do not need to unfold the region led by Line 13. □

Algorithm 1: Unfolding.

```
1  Procedure unfold(𝔾)
2      foreach operator Θ_{κ_i} in 𝔾 do
3          𝔾_forward ← subgraph reachable from but excluding 𝔾_{κ_i} and 𝔾_{¬κ_i};
4          𝕍 ← all vertices including Θ_{κ_i} expressions;
5          𝔾_backward ← subgraph that can reach any vertex in 𝕍, including 𝕍;
6          𝔾_slice ← overlapping subgraph of 𝔾_forward and 𝔾_backward;
7          𝔾'_slice ← a copy of 𝔾_slice, including all its incoming/outgoing edges;
8          disconnect 𝔾_slice from 𝔾_{¬κ_i};
9          replace all Θ_{κ_i} expressions in 𝔾_slice with their first operands;
10         disconnect 𝔾'_slice from 𝔾_{κ_i};
11         replace all Θ_{κ_i} expressions in 𝔾'_slice with their second operands;
```

5.4 Localized Graph Reordering

As illustrated in Figure 4, bytes in a packet may not appear in the order in a program path, e.g., $B[5]$ may precede $B[2]$. To produce legitimate BNF productions, we need to reorder them to produce the ordered AFG. Then transforming an ordered AFG to BNF productions is straightforward. We first define the concepts of *vertical decomposition* (VD) and *horizontal decomposition* (HD).

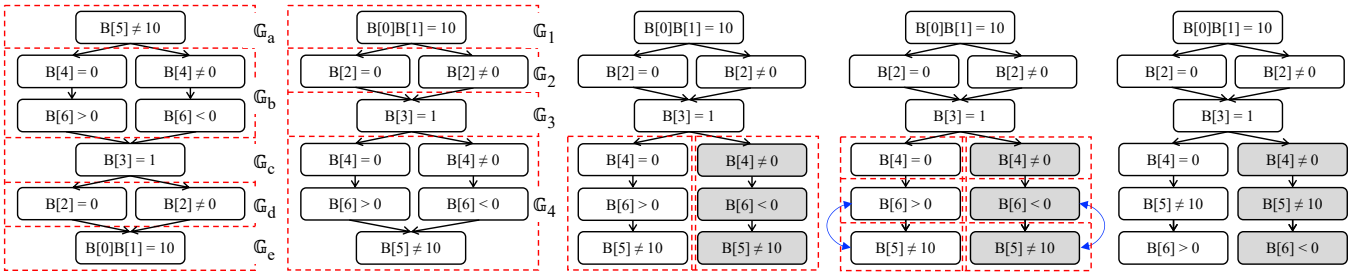
Definition 2 (VD). Given an unfolded AFG $\mathbb{G} = \mathbb{G}_1 \bowtie \mathbb{G}_2 \bowtie \ldots \bowtie \mathbb{G}_n$, namely, the exit vertices in $\mathbb{G}_i$ are fully connected to the entry vertices in $\mathbb{G}_{i+1}$, its vertical decomposition is the sequence of subgraphs, denoted as $\text{VD}(\mathbb{G}) = \mathbb{G}_1\mathbb{G}_2 \ldots \mathbb{G}_n$.

Definition 3 (HD). Given an unfolded AFG $\mathbb{G}$, its horizontal decomposition is a set of subgraphs, each of which is rooted at a single entry vertex in the AFG and includes the subgraph reachable from the entry vertex, denoted as $\text{HD}(\mathbb{G}) = \mathbb{G}_1|\mathbb{G}_2|\ldots|\mathbb{G}_n$.

Figure 10(a) shows an example of vertical decomposition, where the graph is decomposed into two parts, one containing the vertices ρ_1 and ρ_2, and the other containing the vertices ρ_3, ρ_4, and ρ_5. The graph in Figure 10(b) cannot be vertically decomposed because the upper two vertices are not fully connected to the other three. Instead, it can be horizontally decomposed into two parts, one containing the vertices ρ_1, ρ_3, ρ_4, and ρ_5, and the other containing the vertices ρ_2, ρ'_3, and ρ'_5. Here, ρ'_3 and ρ'_5 are copies of ρ_3 and ρ_5, respectively. As illustrated in the example and stated in Lemma 5.4, the AFGs before and after decomposition contain the same number of paths and the constraint represented by each path is not changed.

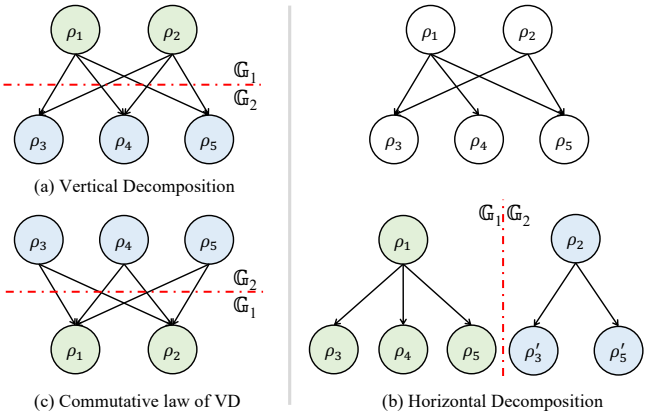
Lemma 5.4. *AFGs before and after decomposition are equivalent in representing path constraint.*

The decomposition has three properties. First, the horizontal decomposition is more expensive than the vertical one as it may copy vertices. Hence, Algorithm 2 always tries the vertical decomposition first. Second, as stated in Lemma 5.5, the decomposition can be recursively performed on a graph and its subgraphs. For instance, after the horizontal decomposition in Figure 10(b), we can further apply vertical decomposition to each subgraph. This property allows us to describe our reordering approach as a recursive process in Algorithm 2. Third, the vertical decomposition follows the commutative law stated in Lemma 5.6. For instance,



(a) VD (b) Reordering & Merging (c) HD (d) VD (e) Result

Figure 9: Example of Algorithm 2.



(a) Vertical Decomposition (b) Horizontal Decomposition

(c) Commutative law of VD

Figure 10: Decomposition for graph reordering.

after switching $\mathbb{G}_1$ and $\mathbb{G}_2$ in Figure 10(a), we get the graph in Figure 10(c), which is equivalent to the original graph because they represent equivalent path constraints: $(\rho_1 \vee \rho_2) \wedge (\rho_3 \vee \rho_4 \vee \rho_5)$ and $(\rho_3 \vee \rho_4 \vee \rho_5) \wedge (\rho_1 \vee \rho_2)$. Such a commutative property allows us to reorder vertices in Algorithm 2.

Lemma 5.5. *If an AFG with multiple vertices cannot be vertically decomposed, each subgraph after horizontal decomposition contains a single vertex or can be vertically decomposed.*

Lemma 5.6. *Switching the position of subgraphs in VD yields an AFG that represents an equivalent constraint as the original AFG.*

Algorithm 2 first tries to vertically decompose the input AFG (Line 2). If failed, Lemma 5.5 allows us to horizontally decompose it into subgraphs and recursively order each subgraph (Lines 14-15). If VD succeeds in splitting AFG into a list of subgraphs, these subgraphs are reordered by byte indices (Lines 3-5). Figure 9(a) and Figure 9(b) illustrate this step. In Figure 9(a), the AFG is vertically decomposed into five subgraphs, $\mathbb{G}_a$, $\mathbb{G}_b$, $\mathbb{G}_c$, $\mathbb{G}_d$, and $\mathbb{G}_e$, which are respectively put in five dashed boxes. The minimum byte indices of the subgraphs are 5, 4, 3, 2, and 0. Figure 9(b) shows the AFG after reordering the subgraphs based on the minimum byte indices. After reordering, since $\mathbb{G}_a$ and $\mathbb{G}_b$ contain overlapping byte indices[1], they are merged into a single subgraph, i.e., $\mathbb{G}_4$ in Figure 9(b). In this example, the subgraphs after reordering and merging are put

[1]The range of byte indices in $\mathbb{G}_a$ is [5, 5], and the range in $\mathbb{G}_b$ is [4, 6]. The former is a subset of the latter. Thus, they overlap each other.

Algorithm 2: Reordering.

```
1  Procedure reorder(𝔾)
2      if VD(𝔾) = 𝔾_a 𝔾_b ... then
3          reorder 𝔾_a, 𝔾_b, ..., based on the min byte index of each subgraph;
4          merge adjacent subgraphs if they contain overlapping byte indices;
5          let 𝒜 ← [𝔾_1, 𝔾_2, ...] be subgraphs after reordering and merging;
6          foreach 𝔾_i ∈ 𝒜 do
7              if 𝔾_i = 𝔾_{i,1} ⋈ 𝔾_{i,2} ⋈ ... is a merged graph then
8                  assume 𝔾_{i,j} has multiple entry vertices;
9                  switch the position of 𝔾_{i,j} and 𝔾_{i,1} in 𝔾_i;
10                 assume HD(𝔾_i) = 𝔾_{a'} | 𝔾_{b'} | ...;
11                 reorder(𝔾_{a'}); reorder(𝔾_{b'}); ...;
12             else
13                 reorder(𝔾_i);
14     else if HD(𝔾) = 𝔾_a | 𝔾_b | ... then
15         reorder(𝔾_a); reorder(𝔾_b); ...;
16     else // a single-vertex graph, do nothing
```

Algorithm 3: Packet Format in BNF.

```
1  Procedure bnf(𝔾)
2      L ← new non-terminal symbol;
3      if VD(𝔾) = 𝔾_a 𝔾_b ... then
4          L → bnf(𝔾_a) bnf(𝔾_b) ...;
5      else if HD(𝔾) = 𝔾_a | 𝔾_b | ... then
6          L → bnf(𝔾_a) | bnf(𝔾_b) | ...;
7      else
           // a single vertex containing B[i], B[i+1], ..., B[i+k]
8          L → B[i]B[i+1]...B[i+k] with assertions in the vertex;
9      return L;
```

in the array $\mathcal{A} = [\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_3, \mathbb{G}_4]$. These subgraphs are ordered and contain mutually exclusive byte indices.

We then recursively reorder subgraphs in $\mathcal{A}$ (Lines 6-13). Especially, for a merged subgraph, e.g., $\mathbb{G}_4 = \mathbb{G}_a \bowtie \mathbb{G}_b$ in the example, since we have tried vertical decomposition, which does not work as neither $\mathbb{G}_a\ \mathbb{G}_b$ nor $\mathbb{G}_b\ \mathbb{G}_a$ respects the stream order, we turn to horizontal decomposition (Lines 8-11). Lines 8-9 ensure the feasibility of horizontal decomposition and Line 10 performs the decomposition. Figure 9(c) illustrates this step, where the subgraph $\mathbb{G}_4$ is horizontally decomposed into the white and the gray parts. Each part then is recursively reordered (Line 11). Figure 9(d) shows that the white and the gray parts are recursively split by vertical decomposition and reordered as indicated by the arrows, yielding the ordered AFG in Figure 9(e). Lemma 5.7 states the correctness of Algorithm 2.

Lemma 5.7. *Algorithm 2 yields an ordered AFG, which represents an equivalent constraint as the input AFG.*

From Ordered AFG to BNF-like Format. It is straightforward to translate an ordered AFG to packet formats in BNF. Due to its simplicity, the detailed discussion is elided and the formal algorithm is put in Algorithm 3. As an example, Figure 4(e) shows the inferred packet format where S is the start symbol that represents the whole graph and each non-terminal L_i represents a subgraph — L_1 represents the path prefix containing $B[0]$, $B[1]$, and $B[2]$; L_2 and L_3 represent two possible constraints of $B[3]$; and L_4 and L_5 stand for the two path suffixes containing $B[4]$, $B[5]$, and $B[6]$.

5.5 Soundness and Completeness in Practice

As proved in Appendix C, Lemmas 5.1-5.7 together guarantee the theoretical soundness and completeness of our approach for a program written in our abstract language. In practice, we need to handle common program structures not included in the abstract language, such as function calls, pointers, and loops. This section discusses how we handle them in our implementation and their effects on soundness or completeness.

Pointers. In the previous discussion, we focus on building an AFG for format inference. Pointer operations are not directly related to AFG. In the implementation, we follow existing works [71] to resolve pointer relations, which helps us identify what values may be loaded from a memory location. For instance, when visiting an assertion in the program such as assert(*(p + 1) > 1) where p is a pointer, if the pointer analysis tells us p+1 points to a memory location storing the value $B[5]$ on the condition ρ, we then compute and include the constraint $\rho \Rightarrow B[5] > 1$ (which equals $\neg\rho \vee B[5] > 1$) in AFG. Pointer operations such as p+1 are not a part of path constraints and, thus, are not included in AFG. That is, according to the assertion rule in Figure 7 and assuming the AFG before the assertion is $\mathbb{G}$, the AFG after the assertion is $\mathbb{G} \bowtie \mathsf{AFG}(\neg\rho \vee B[5] > 1)$. Since the pointer analysis we use is sound and path-sensitive, it allows Netlifter to be sound and highly precise.

Function Calls. Although we do not include function calls in our abstract language for simplicity, our system is inter-procedural as a call statement is equivalent to a list of assignments from the actual parameters to the formals, and a return statement is an assignment from the return value to its receiver. Thus, in our analysis, function calls and returns are treated as assignments. This treatment does not degrade soundness and completeness. Especially, for recursive function calls, we convert them to loops, which are discussed below.

Loops and Repetitive Fields. Loops in a protocol parser are often used to parse repetitive fields [38]. We follow existing techniques to analyze loops [68, 79], which are good at inferring repetitive fields and how many times a field repeats. For example, the code below parses a packet where $B[0]$ represents the packet length and contains a positional constraint that all bytes after $B[0]$ are less than five. For this example, we produce the production $S \rightarrow B[0]B[1]$ with two semantic constraints: $B[1] < 5$ and $\mathsf{repeat}(B[1]) = B[0]$.

```
j = 0;   while(j < packet[0]) { assert(packet[++j] < 5); }
```

Basically, the loop analysis works in two steps. First, they analyze several iterations of a loop. For instance, it analyzes three iterations and gets the results, $j = 1 \wedge B[1] < 5$, $j = 2 \wedge B[2] < 5$, and $j = 3 \wedge B[3] < 5$, for each iteration. Second, it inductively infers the conditions. For instance, the above results can be inductively summarized to $j = k \wedge B[k] < 5$, where $1 \leq k \leq B[0]$ is an induction variable representing the iteration counter. Equivalently, we write the constraint as $B[1] < 5 \wedge \mathsf{repeat}(B[1]) = B[0]$ and, by the definition of AFG, represent it as an edge from a vertex containing $B[1] < 5$ to one containing $\mathsf{repeat}(B[1]) = B[0]$. If the inductive inference succeeds, its result is sound and complete.

```
1.  short compute_crc(char *buf, int len) {
2.      short crc = 0x1d0f;
3.      for (; len > 0; len--) {
4.          crc = (crc >> 8) | (crc << 8);
5.          crc ^= *buf++;                          // read B[k] in (k+1)th iteration
6.          crc ^= (crc & 0xff) >> 4;
7.          crc ^= crc << 12;
8.          crc ^= (crc & 0xff) << 5;
9.      }
10.     return crc;
11. }
12.
13. void parse_packet (char *buf, int len) {        // buf: packet; len: packet length
14.     …
15.     short crc = buf[len - 1] << 8 | buf[len - 2];  // crc: B[length -1]B[length - 2]
16.     assert (crc == compute_crc(buf, len – 2));  // crc == formula(B[0], B[1], B[2], … )
17.     …
18. }
```

Figure 11: An irregular loop that computes checksum.

Irregular Loops. Nevertheless, not all loops can be inductively summarized as above. This is an inherent limitation of static analysis. For irregular loops, we follow the practice of bounded model checking [26] to unroll loops a fixed number of times. While this may introduce unsoundness and incompleteness, we observe few irregular loops (involved in packet parsing) in practice and their influence is limited. In other words, it may produce some unsound or incomplete constraints for some fields in a packet but such unsoundness and incompleteness are rarely propagated to other fields.

The most common irregular loops related to protocol formats are those computing the checksum. Figure 11 shows an example where the function *compute_crc* computes the CRC value as the checksum and the function *parse_packet* compares the computed CRC value to the received one in the packet. The CRC value is computed by a loop, which cannot be inductively summarized. This is because, unlike the variable j in the previous example, we cannot write the value of *crc* as a formula parameterized by the loop counter k.

Thus, we unroll the loop a fixed number of times t, yielding an unsound and incomplete formula of *crc* that relies on $B[0]$, $B[1]$, …, $B[t-1]$, denoted as $f(B[0], B[1], \ldots, B[t-1])$. At Line 16, this formula is compared to the CRC field in the packet. Generally, by the assertion rule in Figure 7, an AFG node with the constraint $B[length-1]B[length-2] = f(B[0], B[1], \ldots, B[t-1])$ should be appended to the AFG. However, in the implementation, we skip such unsound and incomplete constraints, so that they are not included in the protocol format output by Netlifter. Excluding these constraints keeps our inferred format sound but may lose some precision (i.e., completeness) as we miss some constraints.

While we cannot compute precise non-recursive FOL constraints for checksum fields (this is a limitation shared by existing works on protocol format reverse engineering), as discussed in §5.2, we can still infer the boundary of checksum fields. Such field boundaries, together with all sound constraints we compute, can already support many security applications (see §6.3).

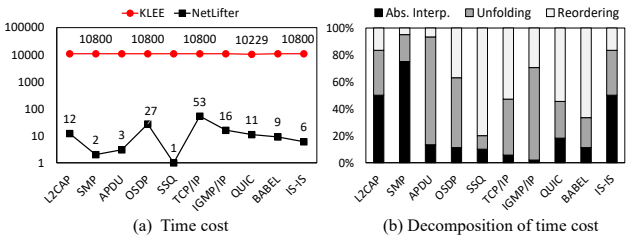
Table 1: Protocols and Their Codebases for Evaluation

Name	Codebase	Size (kloc)	Time (sec.)	Description
L2CAP	linux/bluetooth [10]	38	12	logical link ctrl and adaptation proto.
SMP	linux/bluetooth [10]	12	2	low energy security manager proto.
APDU	opensc [17]	3	3	application proto. data unit
OSDP	libosdp [7]	14	27	open supervised device proto.
SSQ	libssq [11]	8	1	source server query proto.
TCP/IP	lwip [9]	41	53	transport control & internet proto.
IGMP/IP	lwip [9]	17	16	internet group mgmt. & internet proto.
QUIC	ngtcp2 [6]	59	11	general-purpose transport layer proto.
BABEL	frrouting [5]	7	9	a distance-vector routing proto.
IS-IS	frrouting [5]	22	6	intermediate system (IS) to IS proto.
A2MP	linux/bluetooth [10]	16	2	amp manager proto.
BNEP	linux/bluetooth [10]	15	3	BT network encapsulation proto.
CMTP	linux/bluetooth [10]	20	1	c-api message transport proto.
HIDP	linux/bluetooth [10]	17	4	human interface device proto.
UDP	lwip [9]	37	33	user datagram proto.
ICMP	lwip [9]	22	12	internet control message proto.
DHCP	lwip [9]	25	43	dynamic host configuration proto.
ICMP6	lwip [9]	30	54	internet control message proto. v6
DHCP6	lwip [9]	35	51	dynamic host configuration proto. v6
BGP	frrouting [5]	13	2	border gateway proto.
LDP	frrouting [5]	20	5	label distribution proto.
BFD	frrouting [5]	10	17	bidirectional forwarding detection
VRRP	frrouting [5]	8	12	virtual router redundancy proto.
EIGRP	frrouting [5]	14	21	interior gateway routing proto.
NHRP	frrouting [5]	11	11	next hop resolution proto.
OSPF2	frrouting [5]	9	14	open shortest path first v2
OSPF3	frrouting [5]	7	16	open shortest path first v3
RIP1	frrouting [5]	11	13	routing information proto. v1
RIP2	frrouting [5]	11	15	routing information proto. v2
RIPng	frrouting [5]	7	41	routing information proto. for ip6

6 EVALUATION

We implement our method as a tool, namely Netlifter, to lift packet formats from source code in C. It is implemented on top of the LLVM (12.0.0) compiler infrastructure [54] and the Z3 (4.8.12) SMT solver [40]. The source code of a protocol is compiled into the LLVM bitcode, where we perform our static analysis. In the analysis, Z3 is used to represent abstract values as symbolic expressions and compute/solve path constraints. All experiments are run on a Macbook Pro (16-inch, 2019) equipped with an 8-core 16-thread Intel Core i9 CPU with 2.30GHz speed and 32GB of memory.

As shown in Table 1, we have run Netlifter over a number of protocols. They are from different codebases (e.g., Linux and LWIP) and domains (e.g., IoT and routing). They include widely-used ones such as TCP/IP and niche ones like APDU that is used in smart cards. As shown in the table, the size of the code involved in a protocol parser ranges from 3KLoC to 59KLoC, and it takes Netlifter <1min to infer the format of each protocol. Determining the precision and recall of the inferred formats requires manually comparing them with their official documents. We cannot afford to manually inspect all protocols because we have to learn a lot of domain-specific knowledge to understand a protocol, which is time-consuming and not very related to our core contribution to the static analysis. In the remaining experiments, we focus on the first ten, which are from different codebases. We believe that other protocols in the same codebases are implemented in similar manners and, thus, do not introduce extra challenges. We use these protocols/codebases because of two reasons. First, their repositories in GitHub are relatively active, which makes it easy to get feedback from developers when we report bugs. Second, they have their own fuzzing drivers, meaning that they have been extensively fuzzed by the developers themselves. Thus, their code is expected to be of high quality and an approach that can find vulnerabilities in their codebase is highly effective.



(a) Time cost (b) Decomposition of time cost

Figure 12: Time cost (seconds) and its decomposition.

6.1 Effectiveness of the Three-Step Design

For technical contributions, we explained in §4 that our static analysis avoids individually exploring program paths to address two challenges. To show the importance of our solution, we implement a baseline that employs a well-known symbolic executor, KLEE [31], to infer packet formats. Similar to our solution, it infers packets by computing path constraints. Different from our solution, it has to analyze individual program paths. We then compare their time cost of format inference. The results are shown in Figure 12(a) in log scale. The line chart shows that the KLEE-based approach runs out of time ($\geq$ 3 hours) for almost all protocols. We use a three-hour time budget here as it is sufficient to show the advantage of our approach over symbolic execution. As plotted in Figure 12(a), Netlifter can finish in one minute. Figure 12(b) shows the decomposition of Netlifter's time cost, indicating that the three steps of Netlifter respectively take 14%, 44%, and 42% of the total time.

6.2 Precision and Recall of Packet Formats

As discussed in §1, existing techniques focus on network trace analysis (category one) or dynamic program analysis (category two). We refer to both of them as dynamic analyses as they rely on dynamically captured network packets as their inputs. We cannot find any static program analysis that infers formats from a protocol parser. Thus, while the dynamic analyses have a different assumption from our static analysis, not for a comparative purpose but to show the value of our approach, we evaluate Netlifter with two network trace analyses, i.e., NemeSys [50, 51] and NetPlier [80], and two dynamic program analyses, i.e., AutoFormat [58] and Tupni [38]. NemeSys and NetPlier are open-source software and we directly use their implementation. AutoFormat and Tupni are not publicly available. We implement them on top of LLVM based on their papers. We cannot find other open-source dynamic program analyses for evaluation. We evaluate them in terms of precision and recall. Given a set of packets, the precision is the ratio of correctly inferred fields in the packets to all inferred fields. The recall is the ratio of correctly inferred fields to all fields in the ground truth. To compute the precision and recall, we manually build the formats based on the protocols' official documents or source code. We then write scripts to compare the inferred and the manually-built formats.

Dynamic Analysis. To use dynamic analyses, we follow their original works to collect 1000 network packets for each protocol from publicly available datasets [12, 15, 67]. Table 2 shows the precision and recall of the inferred field boundaries. Network trace analyses often exhibit low precision (<50%) and recall (<50%), because they use statistical approaches to align message fields while

Table 2: Precision(%)/Recall(%) of Field Boundaries.

Protocol	Netlifter	NemeSys	NetPlier	AutoFormat	Tupni	Combined
L2CAP	96/98	14/9	14/15	72/32	88/41	66/49
SMP	100/100	27/37	20/52	100/52	96/78	45/82
APDU	100/100	52/21	43/45	44/25	100/61	58/71
OSDP	100/100	17/11	10/16	74/31	89/47	43/52
SSQ	100/100	25/1	26/11	88/19	95/54	41/67
TCP/IP	98/95	5/7	4/12	24/19	88/21	39/25
IGMP/IP	99/98	13/12	13/22	35/22	92/25	54/36
QUIC	97/99	19/14	18/25	70/29	86/43	69/53
BABEL	99/99	28/14	37/8	43/16	80/24	40/28
IS-IS	98/99	23/5	19/14	100/34	87/21	62/42

Table 3: Precision(%)/Recall(%) of Field Names.

L2CAP	SMP	APDU	OSDP	SSQ
100/97	100/96	100/100	100/100	100/100
TCP/IP	**IGMP/IP**	**QUIC**	**BABEL**	**IS-IS**
100/95	100/98	96/96	100/95	100/94

statistical approaches are known to have inherent uncertainty and their effectiveness heavily hinges on the quality of input packets.

The two dynamic program analyses, especially Tupni, significantly improve the precision due to the analysis of control/data flows in the code. AutoFormat has a relatively low precision because it tracks coarse-grained control/data flows. For instance, AutoFormat regards consecutive bytes of a packet processed in the same calling context as a single field. However, it is common for a parser to process multiple fields in the same calling context. Tupni tracks more fine-grained control/data flows, such as predicates in the code, and, thus, exhibits a higher precision. As acknowledged by Tupni itself, it may also produce false fields in many cases. For instance, when the value of a multi-byte field is computed by multiple instructions over every single byte in the field, it will incorrectly split the field into multiple fields. Despite the high precision achieved by Tupni, the key problem of these dynamic analyses is their coverage (i.e., recall), which is often lower than 50% and may compromise downstream security analyses as discussed in the next subsection.

Note that simply combining the results of multiple tools does not help improve the quality of the inferred formats. This is because, when combining the formats inferred by multiple tools, with the increase of correctly inferred fields, the number of incorrect fields also increases. For instance, after combining the results of the four dynamic tools, the precision for OSDP is 0.43, which is even worse than the result when using Tupni independently. The combined results are shown in the last column of Table 2.

Static Analysis. Table 2 shows that, in terms of field boundaries, our inferred formats cover >96% formats and produce <4% false ones. For many of them, we can produce absolutely correct formats. We also miss some fields and report some false ones due to the inherent limitations of static analysis (see §9). These limitations, e.g., the incapability of handling inline-assembly in the source code, will let us lose information during the static analysis, thereby leading to false formats. Table 3 also shows the quality of the inferred field names. A name is considered to be correct if it is the same as the official documents or a reasonable abbreviation, e.g., 'len' vs. 'length'. Overall, we can infer >94% field names with a precision >96%. The names provide high-level semantics and help us identify special fields to facilitate security applications as discussed next.

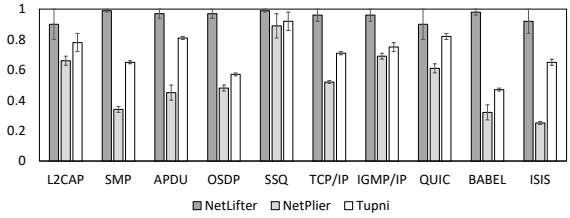


Figure 13: The y-axis is the number of covered branches normalized to one. It shows the branch coverage averaged over twenty runs with a 95% confidence interval.

6.3 Security Applications

Protocol Fuzzing. To show the value of our approach, we respectively input the formats inferred by Netlifter, NetPlier, and Tupni to a typical grammar-based (i.e., format-based) protocol fuzzer, namely BooFuzz [14, 16]. Particularly, since we can locate checksum fields by names such as 'checksum' and 'crc', in the fuzzing experiments, we can skip the checksum checks in the code. This is critical for fuzzing as random mutations in fuzzing can easily invalidate the checksum values [73]. The experiments are performed on a three-hour budget and repeated twenty times to avoid random factors. We use a three-hour budget because we observe that the baseline fuzzers rarely achieve new coverage after three hours.

The results are shown in Figure 13. Since Netlifter can provide formats with precise field boundaries and semantic constraints, Netlifter-enhanced BooFuzz achieves 1.2× to 3.6× coverage compared to others. Netlifter-enhanced BooFuzz also detected 53 zero-day vulnerabilities while the others detect only 12. All detected vulnerabilities are exploitable as they can be triggered via crafted network packets. To date, 47 of them have been assigned CVEs. We can detect more bugs as our inferred formats are of both high precision and high coverage. In Appendix A, we provide more details about the fuzzing experiments and the detected bugs.

Traffic Auditing and Intrusion Detection. Appendix B provides an extended study, where we use the formats inferred by Netlifter and the best baseline, Tupni, to enhance Wireshark and Snort. We conclude that the precise and high-coverage formats inferred by us are critical for auditing traffic and detecting intrusions.

7 RELATED WORK

Existing techniques that infer packet formats are mainly based on dynamic analysis. We discuss some typical ones in what follows. For a broader overview, we refer readers to four surveys [44, 56, 65, 70].

Network Trace Analysis (NTA). NTA uses statistical methods to identify field boundaries based on runtime network packets. Discoverer [36] relies on a recursive clustering approach to recognize packets of the same type. Biprominer [74] uses the variable length pattern to locate protocol keywords and is enhanced by ProDecoder [75]. AutoReEngine [61] uses data mining to identify protocol keywords, based on which packets are classified into different types. ReverX [23] uses a speech recognition algorithm to identify delimiters in packets. NemeSys [50, 51] interprets binary packets as feature vectors and applies an alignment and clustering algorithm to determine the packet format. NetPlier [80] leverages a probabilistic analysis to determine the keyword field, clusters

packets based on the keyword values, and applies multi-sequence alignment to derive packet format. These techniques do not analyze code and, thus, are different from ours.

Dynamic Program Analysis (DPA). DPA can be used over both source and binary code. They work by running protocol parsers against network packets and monitoring runtime control/data flows. Polyglot [30] uses dynamic taint analysis to infer fixed or variable length fields. AutoFormat [58] approximates the field hierarchical structure by monitoring call stacks. This approach then is extended to both bottom-up and top-down hierarchical structures [59]. Wondracek et al. [78] identify delimiters and length fields within a hierarchical structure. Tupni [38] tracks fine-grained taint flows to identify packet fields. It also applies loop analysis to infer repeated fields and records path constraints to infer length or checksum fields. ReFormat [77] recognizes encrypted fields based on the observation that encrypted fields are processed by a high percentage of arithmetic/bitwise instructions. Our approach can be easily extended with the same observation, i.e., by counting relevant instructions to recognize an encrypted field. In addition to inferring the formats of received packets, Dispatcher [29] and P2C [52] reverse engineer the formats of packets to be sent and, thus, are different from all aforementioned approaches as well as ours.

Static Program Analysis (SPA). There are a few SPAs for reverse engineering protocols. However, they either infer the formats of packets to be sent via imprecise abstract domain [57] or focus on cryptographic mechanisms [24]. Our approach precisely infers the format of received packets and, thus, is different from these works.

8 CONCLUSION

In this work, we propose a static analysis that can infer protocol formats with both high precision and high recall. Hence, the formats significantly enhance network protocol fuzzing, network traffic auditing, and network intrusion detection. Particularly, our format-inference technique has helped existing protocol fuzzers find 53 zero-days with 47 assigned CVEs.

9 LIMITATIONS AND FUTURE WORK

Our static analysis currently is implemented for C and does not support C++ due to the difficulty in analyzing virtual tables. We focus on the source code and do not handle inline assembly and libraries that do not have code available. We believe these limitations can be addressed with more engineering work. For instance, we can use class hierarchical analysis, e.g., [42], to deal with virtual tables and support C++. We can use existing disassembly techniques, e.g., [63], to support inline assembly. We leave them as our future work.

As discussed earlier, Netlifter employs existing techniques to deal with pointers and loops. Thus, it inherits their limitations. A common limitation shared by both Netlifter and all recent techniques is that the quality of inferred formats relies on the protocol implementation. For instance, if the implementation ignores a field, the output formats will ignore it, either. Nevertheless, we have shown that Netlifter is promising in practice via a set of experiments.

REFERENCES

[1] 2002. ISO/IEC 10589:2002. https://www.iso.org/standard/30932.html. (2002).
[2] 2016. ISIS PDU format. https://techhub.hpe.com/eginfolib/networking/docs/switches/3600v2/5998-7619r_l3-ip-rtng_cg/content/442284234.htm. (2016).
[3] 2018. 2018 IBM X-Force Report. https://securityintelligence.com/2018-ibm-x-force-report-shellshock-fades-gozi-rises-and-insider-threats-soar/. (2018).
[4] 2022. Documents of OSDP. https://libosdp.gotomain.io/. (2022).
[5] 2022. The FRRouting protocol suite. https://github.com/FRRouting/frr. (2022).
[6] 2022. The IETF QUIC protocol. https://github.com/ngtcp2/ngtcp2. (2022).
[7] 2022. Implementation of OSDP. https://github.com/goToMain/libosdp. (2022).
[8] 2022. IS-IS. https://en.wikipedia.org/wiki/IS-IS. (2022).
[9] 2022. A lightweight TCPIP stack. https://github.com/lwip-tcpip/lwip. (2022).
[10] 2022. Linux kernel source tree. https://github.com/torvalds/linux. (2022).
[11] 2022. A modern source server query protocol library written in C. https://github.com/BinaryAlien/libssq. (2022).
[12] 2022. Network forensics tools and datasets. https://github.com/MartinaZembjakova/Network-forensic-tools-taxonomy. (2022).
[13] 2022. Network intrution & prevention systems. https://www.snort.org/. (2022).
[14] 2022. Network protocol fuzzing. https://github.com/jtpereyda/boofuzz. (2022).
[15] 2022. Packet captures. https://packetlife.net/captures. (2022).
[16] 2022. A pure-python fully automated and unattended fuzzing framework. https://github.com/OpenRCE/sulley. (2022).
[17] 2022. Smart card tools. https://github.com/OpenSC/OpenSC. (2022).
[18] 2022. Wireshark. https://www.wireshark.org//. (2022).
[19] 2023. American fuzzy lop. https://lcamtuf.coredump.cx/afl/. (2023).
[20] 2023. Lifting network implementation to precise format specification. https://github.com/qingkaishi/netlifter. (2023).
[21] 2023. Top-down parsing. https://wikipedia.org/wiki/Top-down_parsing. (2023).
[22] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: Principles, Techniques, and Tools.* Pearson Addison Wesley.
[23] Joao Antunes, Nuno Neves, and Paulo Verissimo. 2011. Reverse engineering of protocols from network traces. In *Working Conference on Reverse Engineering (WCRE '11)*. IEEE, 169–178.
[24] Matteo Avalle, Alfredo Pironti, and Riccardo Sisto. 2014. Formal verification of security protocol implementations: a survey. *Formal Aspects of Computing* 26, 1 (2014), 99–123.
[25] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. ACM, 95–110.
[26] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*. Springer, 193–207.
[27] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. 2005. Rigorous Specification and Conformance Testing Techniques for Network Protocols, as Applied to TCP, UDP, and Sockets. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '05)*. ACM, 265–276.
[28] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. 2006. Engineering with Logic: HOL Specification and Symbolic-Evaluation Testing for TCP Implementations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, 55–66.
[29] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. 2009. Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-Engineering. In *ACM Conference on Computer and Communications Security (CCS '09)*. ACM, 621–634.
[30] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In *ACM Conference on Computer and Communications Security (CCS '07)*. ACM, 317–329.
[31] Cristian Cadar, Daniel Dunbar, Dawson R. Engler, et al. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*. USENIX, 209–224.
[32] Chia Yuan Cho, Domagoj Babić, Eui Chul Richard Shin, and Dawn Song. 2010. Inference and Analysis of Formal Models of Botnet Command and Control Protocols. In *ACM Conference on Computer and Communications Security (CCS '10)*. ACM, 426–439.
[33] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. 2009. Prospex: Protocol specification extraction. In *2009 30th IEEE Symposium on Security and Privacy (S&P '09)*. IEEE, 110–125.
[34] Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '79)*. ACM, 269–282.
[35] Dave Crocker and Paul Overell. 1997. *Augmented BNF for syntax specifications: ABNF.* Technical Report. RFC 2234.
[36] Weidong Cui, Jayanthkumar Kannan, and Helen Wang. 2007. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium (Security '07)*. USENIX, 199–212.

[37] Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy H Katz. 2006. Protocol-independent adaptive replay of application dialog.. In *Network and Distributed System Security Symposium (NDSS '06)*. Internet Society, 1–15.

[38] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. 2008. Tupni: Automatic Reverse Engineering of Input Formats. In *ACM Conference on Computer and Communications Security (CCS '08)*. ACM, 391–402.

[39] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1989. An efficient method of computing static single assignment form. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. ACM, 25–35.

[40] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*. Springer, 337–340.

[41] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *USENIX Security Symposium (Security '15)*. USENIX, 193–206.

[42] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95)*. Springer, 77–101.

[43] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: Safe Asynchronous Event-Driven Programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, 321–332.

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

[45] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems (SecureComm '15)*. Springer, 330–347.

[46] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining Input Grammars from Dynamic Control Flow. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*. ACM, 172–183.

[47] Matthias Höschele and Andreas Zeller. 2016. Mining input grammars from dynamic taints. In *International Conference on Automated Software Engineering (ASE '16)*. ACM, 720–725.

[48] Bahruz Jabiyev, Steven Sprecher, Anthony Gavazzi, Tommaso Innocenti, Kaan Onarlioglu, and Engin Kirda. 2022. FRAMESHIFTER: Security Implications of HTTP/2-to-HTTP/1 Conversion Anomalies. In *USENIX Security Symposium (Security '22)*. USENIX, 1061–1075.

[49] James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[50] Stephan Kleber, Henning Kopp, and Frank Kargl. 2018. NEMESYS: Network message syntax reverse engineering by analysis of the intrinsic structure of individual messages. In *USENIX Workshop on Offensive Technologies (WOOT '18)*. USENIX, 1–13.

[51] Stephan Kleber, Rens W. van der Heijden, and Frank Kargl. 2020. Message Type Identification of Binary Network Protocols using Continuous Segment Similarity. In *IEEE Conference on Computer Communications (INFOCOM '20)*. IEEE, 2243–2252.

[52] Yonghwi Kwon, Fei Peng, Dohyeong Kim, Kyungtae Kim, Xiangyu Zhang, Dongyan Xu, Vinod Yegneswaran, and John Qian. 2015. P2C: Understanding output data files via on-the-fly transformation from producer to consumer executions. In *Network and Distributed System Security Symposium (NDSS '15)*. Internet Society, 1–14.

[53] Patrick LaRoche, A Nur Zincir-Heywood, and Malcolm I Heywood. 2012. Network protocol discovery and analysis via live interaction. In *European Conference on the Applications of Evolutionary Computation (ECAEC '12)*. Springer, 11–20.

[54] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization (CGO '04)*. IEEE, 75:1–75:12.

[55] Corrado Leita, Ken Mermoud, and Marc Dacier. 2005. ScriptGen: an automated script generation tool for Honeyd. In *Annual Computer Security Applications Conference (ACSAC '05)*. IEEE, 203–214.

[56] Xiangdong Li and Li Chen. 2011. A survey on methods of automatic protocol reverse engineering. In *Proceedings of the 7th International Conference on Computational Intelligence and Security (CIS '11)*. IEEE, 685–689.

[57] Junghee Lim, Thomas Reps, and Ben Liblit. 2006. Extracting output formats from executables. In *Working Conference on Reverse Engineering (WCRE '06)*. IEEE, 167–178.

[58] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. 2008. Automatic protocol format reverse engineering through context-aware monitored execution. In *Network and Distributed System Security Symposium (NDSS '08)*. Internet Society, 1–15.

[59] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Reverse engineering input syntactic structure from program execution and its applications. *IEEE Transactions on Software Engineering* 36, 5 (2010), 688–703.

[60] Min Liu, Chunfu Jia, Lu Liu, and Zhi Wang. 2013. Extracting sent message formats from executables using backward slicing. In *Proceedings of the 4th International Conference on Emerging Intelligent Data and Web Technologies (EIDWT '13)*. IEEE, 377–384.

[61] Jian-Zhen Luo and Shun-Zheng Yu. 2013. Position-based automatic reverse engineering of network protocols. *Journal of Network and Computer Applications* 36, 3 (2013), 1070–1077.

[62] Chris McMahon Stone, Sam L. Thomas, Mathy Vanhoef, James Henderson, Nicolas Bailluet, and Tom Chothia. 2022. The Closer You Look, The More You Learn: A Grey-Box Approach to Protocol State Machine Learning. In *ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. ACM, 2265–2278.

[63] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic disassembly. In *International Conference on Software Engineering (ICSE '19)*. IEEE, 1187–1198.

[64] Madanlal Musuvathi, Dawson R Engler, et al. 2004. Model Checking Large Network Protocol Implementations. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*. USENIX, 1–14.

[65] John Narayan, Sandeep Shukla, and Charles Clancy. 2015. A survey of automatic protocol reverse engineering tools. *Comput. Surveys* 48, 3 (2015), 1–26.

[66] P. Oehlert. 2005. Violating assumptions with fuzzing. *IEEE Security & Privacy* 3, 2 (2005), 58–62.

[67] Markus Ring, Sarah Wunderlich, Deniz Scheuring, Dieter Landes, and Andreas Hotho. 2019. A survey of network-based intrusion detection data sets. *Computers & Security* 86, 1 (2019), 147–167.

[68] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. 2009. Loop-extended symbolic execution on binary programs. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '09)*. ACM, 225–236.

[69] Maxim Shevertalov and Spiros Mancoridis. 2007. A Reverse Engineering Tool for Extracting Protocols of Networked Applications. In *Working Conference on Reverse Engineering (WCRE '07)*. IEEE, 229–238.

[70] Baraka D Sija, Young-Hoon Goo, Kyu-Seok Shim, Huru Hasanova, and Myung-Sup Kim. 2018. A survey of automatic protocol reverse engineering approaches, methods, and tools on the inputs and outputs view. *Security and Communication Networks* 2018, 8370341 (2018), 1–17.

[71] Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew. 2011. SPAS: Scalable path-sensitive pointer analysis on full-sparse SSA. In *Proceedings of the 9th Asian Symposium on Programming Languages and Systems (APLAS '11)*. Springer, 155–171.

[72] Fei Wang, Jianliang Wu, Yuhong Nan, Yousra Aafer, Xiangyu Zhang, Dongyan Xu, and Mathias Payer. 2022. ProFactory: Improving IoT Security via Formalized Protocol Customization. In *USENIX Security Symposium (Security '22)*. USENIX, 1–18.

[73] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy (S&P '10)*. IEEE, 497–512.

[74] Yipeng Wang, Xingjian Li, Jiao Meng, Yong Zhao, Zhibin Zhang, and Li Guo. 2011. Biprominer: Automatic mining of binary protocol features. In *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT '11)*. IEEE, 179–184.

[75] Yipeng Wang, Xiaochun Yun, M Zubair Shafiq, Liyan Wang, Alex X Liu, Zhibin Zhang, Danfeng Yao, Yongzheng Zhang, and Li Guo. 2012. A semantics aware approach to automated reverse engineering unknown protocols. In *IEEE International Conference on Network Protocols (ICNP '12)*. IEEE, 1–10.

[76] Yipeng Wang, Zhibin Zhang, Danfeng Daphne Yao, Buyun Qu, and Li Guo. 2011. Inferring protocol state machine from network traces: a probabilistic approach. In *International Conference on Applied Cryptography and Network Security (ACNS '11)*. Springer, 1–18.

[77] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. 2009. ReFormat: Automatic reverse engineering of encrypted messages. In *European Symposium on Research in Computer Security (ESORICS '09)*. Springer, 200–215.

[78] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, and Scuola Superiore S Anna. 2008. Automatic network protocol analysis. In *Network and Distributed System Security Symposium (NDSS '08)*. Internet Society, 1–18.

[79] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. 2016. Proteus: Computing disjunctive loop summary via path dependency analysis. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE '16)*. ACM, 61–72.

[80] Yapeng Ye, Zhuo Zhang, Fei Wang, Xiangyu Zhang, and Dongyan Xu. 2021. NetPlier: Probabilistic network protocol reverse engineering from message traces. In *Symposium on Network and Distributed System Security (NDSS '21)*. Internet Society, 1–18.

[81] Zhao Zhang, Qiao-Yan Wen, and Wen Tang. 2012. Mining Protocol State Machines by Interactive Grammar Inference. In *International Conference on Digital Manufacturing & Automation (ICDMA '12)*. IEEE, 524–527.

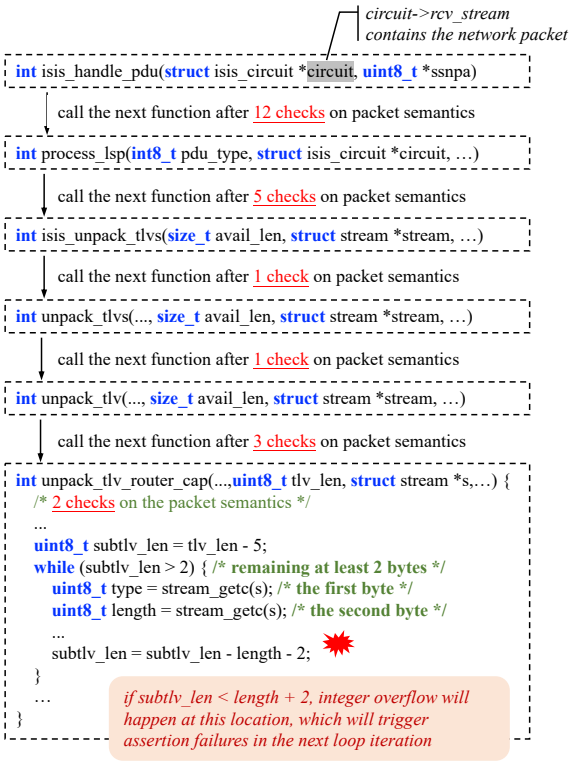


Figure 14: A vulnerability in the code of the IS-IS protocol.

A DETAILS OF THE FUZZING EXPERIMENT

Table 4 shows the breakdown of the detected bugs by the bug types, the protocols, and the detectors. The bugs we detected include integer overflow, buffer overflow, calling invalid addresses, and infinite loops. We detected 53 bugs in total, while the baselines only detect 12 of them.

Table 4: # Bugs Detected by Our Fuzzer (# by Baselines)

Protocol	Integer Overflow	Buffer Overflow	Calling Invalid Addr	Infinite Loops
OSDP		2 (0)	4 (0)	
SSQ		2 (0)		
BABEL		12 (4)		3 (0)
ISIS	22 (8)	8 (0)		

Figure 14 demonstrates the details of a vulnerability we found in IS-IS (Intermediate System to Intermediate System), a widely-used routing protocol. To perform security analysis like fuzzing, we need its format to generate valid IS-IS packets. While it is easy to find some documents of this protocol on the internet, e.g., [1, 2, 8], all of them are written in a natural language, which cannot be directly processed by machines for automatic packet generation. Apparently, manually translating these documents to a machine-readable formal language is labor-intensive and error-prone. Since its implementation is available in GitHub [5], we can use our static analysis to produce its format in a formal language and, hence, facilitate the downstream automated security analysis.

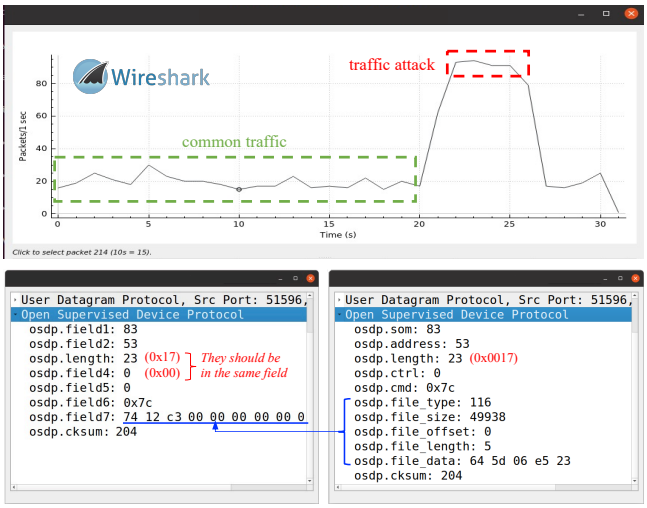
The vulnerability we study here is identified by CVE-2022-26125. Attackers may use it for DoS attacks. This vulnerability has been fixed by the developers of this protocol. Thus, we do not think there are ethical concerns to discuss its details here. As shown in Figure 14, this vulnerability spans over at least six levels of function call, from the function *isis_handle_pdu*, the entry function of the protocol parser, to the function *unpack_tlv_router_cap*, which parses a segment of the network packet. Before the function call at every level, there are at least one and up to twelve conditional statements that check if certain semantic constraints are satisfied. Ideally, these checks are sufficient to prune exploit packets. In total, before reaching the vulnerable location, an exploit packet needs to pass 24 checks of the semantic constraints, which makes it hard for a fuzzer to generate such a packet via random mutation. Hence, a format that allows us to generate bug-triggering network packets must precisely model such semantic constraints and, at the same time, cannot miss packet formats that are qualified to reach the vulnerable code. Our static analysis produces IS-IS formats with both high recall and high precision, thereby allowing us to produce bug-triggering network packets easily. By contrast, on one hand, the format generated by NetPlier contains few semantic constraints. Hence, packets produced based on the format usually violate the semantic constraints and, thus, are easily filtered out by the parser. Hence, these packets cannot execute deep program paths and are hard to trigger the vulnerability. On the other hand, while the format generated by Tupni is much more precise, its recall is only 21%, missing the format of bug-triggering packets. Hence, the fuzzer enhanced by Tupni does not discover this vulnerability, either.

As shown in Figure 14, the vulnerability happens in the function *unpack_tlv_router_cap*, which parses a segment of the input network packet. In the code, the variable *subtlv_len* means the remaining bytes that have not been parsed in the segment. The while loop can only be reached when $subtlv\_len > 2$, i.e., at least two bytes have not been parsed. In the loop, we read two bytes, one to the variable *type* and the other to the variable *length*. The loop then parses *length* bytes. Thus, it parses $length + 2$ bytes in total in each loop iteration. At the end of an iteration, it updates the remaining bytes by subtracting the bytes parsed in the loop iteration from the variable *subtlv_len*. Ideally, the remaining bytes, *subtlv_len*, should always decrease, until $subtlv\_len \leq 0$. However, in the code, *subtlv_len* is an unsigned 8-bit integer and, thus, always positive. If $subtlv\_len < length + 2$, e.g., $subtlv\_len = 35$ and $length + 2 = 36$, the subtraction will not produce an negative integer, -1, but a large positive integer, 255, due to integer overflow. This integer overflow will further lead to assertion failures in the next loop iteration as the loop expects $subtlv\_len = 255$ remaining bytes, which do not exist. Observe that IS-IS is a routing protocol. This vulnerability could lead to DoS attacks when attackers send exploit packets to trigger the vulnerability and crash the routers. If so, legitimate users depending on the routers will not be able to access information systems, devices, or other network resources.

B CASE STUDY VIA WIRESHARK AND SNORT

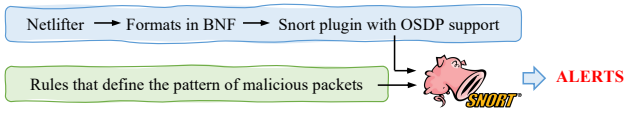
This appendix extends our discussion in §2 to detail the attack model as well as comparing the effectiveness of using Netlifter and Tupni to enhance Wireshark and Snort.

Attack Model. The attack model contains a set of smart-home devices that communicate with a target using OSDP and other protocols. One of the devices, of which the address is 0x35 as shown



(a) Tupni (b) Netlifter

Figure 16: Screenshorts of the enhanced Wireshark.



alert osdp any any -> any any { msg: "malicious osdp packets detected!";
content: "osdp.address = 0x35 && osdp.cmd = 0x7C"; }

Figure 17: Rule-based intrusion detection via Snort [13].

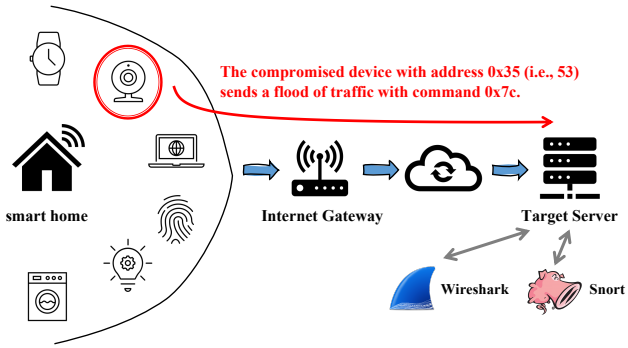


Figure 15: Attack model.

in Figure 15 is compromised to launch a traffic attack, i.e., send a flood of OSDP traffic with the command 0x7C to overwhelm the target. The target runs Wireshark and Snort, two foremost network security analyzers, to audit network traffic and detect network intrusions. However, since the vanilla Wireshark and Snort do not support OSDP, we fail to ensure network security with them.

To support OSDP, we use Netlifter and Tupni to infer the packet formats from the protocol's implementation. Based on the formats, we generate plugins to enhance Wireshark and Snort. Ideally, the enhanced Wireshark and Snort can parse OSDP packets, dissect each packet into multiple fields, and further facilitate the analysis and detection of the traffic attack.

Auditing Abnormal Traffic via Wireshark. We have shown in Figure 2 that the vanilla Wireshark does not support OSDP. Thus, all OSDP packets are shown as raw bytes by Wireshark. In this case study, we further compare Netlifter-enhanced Wireshark and Tupni-enhanced Wireshark. Figure 16 shows the comparison results, which demonstrate three problems of using dynamic analysis for format inference. First, as a dynamic analysis, Tupni is of relatively low coverage and misses many formats. As an example, in Figure 16(a), the bytes in the 7th field are not successfully decoded and, thus, are shown as raw bytes. Second, it mistakenly recognizes some fields. For instance, the length field should contain two bytes but is split into two independent fields, i.e., *length* and *field4* in Figure 16(a), by Tupni (note that the packet length in the ground truth should be computed as $filed4 \times 256 + length$). Third, it does not infer the names of most fields, making it hard to understand. In comparison, our static analysis is of both high precision and high coverage and, meanwhile, infers the name of all fields. Thus, Netlifter-enhanced Wireshark successfully dissects all received packets into fields, provides a proper name for each field, and, thus, effectively helps users to understand the network traffic.

Detecting Intrusion via Snort. Snort is the foremost open-source network intrusion detection system developed by Cisco [13]. It allows users to write rules to define malicious packet patterns. It then finds packets that match the patterns and generates alerts. Figure 17 shows a typical rule. The keyword *alert* indicates the action when a malicious packet is received, *any* means any source/destination ip/port of the network traffic, *msg* defines the warning message when a malicious packet is detected, and *content* defines the pattern of malicious packets, which can be determined by inspecting the attack packets (using Wireshark). Like the Wireshark extension, we develop an extension for Snort based on the lifted protocol formats. The extension basically parses a OSDP packet, dissects it to multiple fields, and checks the Snort rules according to the field values.

During the attack, we can use Wireshark to understand the pattern of received packets. We then write a Snort rule as below to define the pattern of malicious packets and generate alerts for users when receiving them or directly prevent such packets.

alert osdp any any → any any { msg : "malicious packets detected";
content : "osdp.address=53 && osdp.cmd=0x7c"; }

When using Tupni-enhanced Wireshark, we may misunderstand the packets in the traffic attack and define an imprecise pattern, thereby missing the chance of detecting intrusions. As shown in Figure 16(a), Tupni incorrectly splits the two bytes that represent length into two independent fields, *length* and *field4* (note that the packet length in the ground truth should be computed as $filed4 \times 256 + length$). During the analysis, we find *filed4* is always zero because all packets during the attack have a length less than 256. Hence, we will define a pattern with osdp.field4 = 0 as below, which over-constrains the malicious packets:

alert osdp any any → any any { msg : "malicious packets detected";
content : "osdp.field2=53 && osdp.field4=0 && osdp.field6=0x7c"; }

Next time, when attackers send packets longer than 256 bytes, the value of *field4* will no longer be zero and Snort will not be able to prevent such attacks using the rule. In our case study, when using Tupni, Snort misses over 50% of malicious packets while Netlifter-enhanced Snort prevents all malicious packets.

One may think that, to prevent a traffic attack, we can simply block the IP address without looking into the OSDP packets. In fact, this does not work in many cases. For example, on one hand, we may not want to block all traffic from an IP but just block some compromised functionality, e.g., the command 0x7c. In this case, blocking the IP will overkill all functionality of the smart home devices. On the other hand, the IP addresses are often dynamically allocated, blocking a single IP may not work when it is changed. However, in the smart home scenario, the OSDP address of each device can be set statically and, thus, can be reliably used.

C PROOFS FOR THEORETICAL SOUNDNESS & COMPLETENESS

Our approach infers the message formats by building, unfolding, and reordering AFG. Thus, the theoretical soundness and completeness of our approach is proved in the following three steps:

(1) Lemma 5.1 proves that AFG is an equivalent representation of path constraint. Lemma 5.2 shows that, given a program in our abstract language, the path constraint represented by the resulting AFG is sound and complete.
(2) Lemma 5.3 states that the unfolding step does not affect the soundness and completeness of the AFG.
(3) Lemma 5.4, Lemma 5.5, and Lemma 5.6 prove three properties that hold when reordering the AFG. Based on these properties, Lemma 5.7 states that the reordering step does not affect the soundness and completeness of the AFG.

Lemma 5.1: *Given AFG(ρ) with n paths, we have $\rho \equiv \bigvee_{i=1}^{n} \rho_i$ where each ρ_i equals the conjunction of all constraints in an AFG path.*

Proof. In the proof, given any constraint ρ, we use ρ_i to represent the conjunction of all constraints in a path of AFG(ρ).

Base: When a constraint ρ is an atomic constraint without any connectives $\wedge$ or $\vee$, AFG(ρ) returns a single vertex containing ρ. It is apparent that the lemma is correct in this trivial case.

Induction: Consider two constraints, γ and σ as well as their corresponding AFG(γ) and AFG(σ), which contains m and n paths, respectively. Let us assume that the lemma to prove is correct. That is, we have $\gamma \equiv \vee_{i=1}^{m} \gamma_i$ and $\sigma \equiv \vee_{i=1}^{n} \sigma_i$.

Induction Case (1): Consider the constraint $\gamma \vee \sigma$, denoted as ρ. We have AFG(ρ) = AFG(γ) $\uplus$ AFG(σ), which, by definition, consists of two independent subgraphs AFG(γ) and AFG(σ) and, thus, contains and only contains $m + n$ paths from AFG(γ)and AFG(σ). Thus, we have

$$\rho \equiv \gamma \vee \sigma \equiv \bigvee_{i=1}^{m} \gamma_i \vee \bigvee_{i=1}^{n} \sigma_i \equiv \bigvee_{i=1}^{m+n} \rho_i, \textbf{ where } \rho_i = \begin{cases} \gamma_i, & i \leq m \\ \sigma_{i-m}, & i > m \end{cases}$$

Thus, if the lemma is correct for γ and σ, it is also correct for $\gamma \vee \sigma$.

Induction Case (2): Consider the constraint $\gamma \wedge \sigma$, denoted as ρ. we have AFG(ρ) = AFG(γ) $\bowtie$ AFG(σ). In the graph AFG(ρ), all exiting vertices of the subgraph AFG(γ) are connected to all entry vertices of the subgraph AFG(σ). Hence, AFG(ρ) contains $m \times n$ paths, and $\rho_i = \gamma_j \wedge \sigma_k$, where $1 \leq j \leq m$ and $1 \leq k \leq n$. Therefore, we have

$$\rho \equiv \gamma \wedge \sigma \equiv \bigvee_{i=1}^{m} \gamma_i \wedge \bigvee_{i=1}^{n} \sigma_i \equiv \bigvee_{i=1}^{m \times n} \rho_i.$$

Thus, if the lemma is correct for γ and σ, it is also correct for $\gamma \vee \sigma$.

Summary: if the lemma to prove is correct for γ and σ, it is also correct for $\gamma \wedge \sigma$ and $\gamma \vee \sigma$. Thus the lemma to prove is correct. □

The lemma above proves the equivalence relation between the AFG, AFG(ρ), and the constraint ρ. That is, we can always compute the constraint it represents, i.e., ρ, by computing $\bigvee_i \rho_i$, where ρ_i equals the conjunction of all constraints in an AFG path.

Lemma 5.2: *Given a program in the language defined in Figure 5, the AFG produced by the abstract interpretation is sound and complete.*

Proof. The inference rules of the abstract interpretation are shown in Figure 7. For each statement in our abstract language, there is an inference rule that models its exact semantics and does not introduce any over- or under-approximation into the resulting abstract values and AFGs.

For abstract values, as an example, given the abstract values $\tilde{v_2}$ and $\tilde{v_3}$ of the variables v_2 and v_3, the inference rule for the binary operation $v_1 \leftarrow v_2 \oplus v_3$ yields the abstract value $\tilde{v_2} \oplus \tilde{v_3}$ for the variable v_1. This procedure does not introduce any over- or under-approximation. Thus, the abstract values computed by the inference rules are sound and complete.

For AFGs, as an example, given an assertion **assert**(v) in the code and the AFG before the assertion, e.g., AFG(ρ), the assertion rule yields a new AFG, AFG(ρ) $\bowtie$ AFG($\tilde{v}$), which equals AFG($\rho \wedge \tilde{v}$) by the definition of AFG. By Lemma 5.1, the graphs AFG(ρ), AFG($\tilde{v}$), and AFG($\rho \wedge \tilde{v}$) represent the constraints ρ, $\tilde{v}$, and $\rho \wedge \tilde{v}$, respectively. This means that the path constraint before the assertion is ρ and the path constraint after the assertion is $\rho \wedge \tilde{v}$. Apparently, Since the resulting path constraint follows the definition of path constraint and the abstract value $\tilde{v}$ is sound and complete, this rule does not introduce any over- or under-approximation into the resulting path constraint $\rho \wedge \tilde{v}$ and its equivalent representation AFG($\rho \wedge \tilde{v}$).

To sum up, given a program written in our abstract language, since each inference rule does not introduce any over- or under-approximation into the abstract values and AFGs, the path constraint represented by the resulting AFG is sound and complete. □

Note that the proof above assumes that a program is written in our abstract language, which is loop-free. Thus, the static analysis always converges with a sound and complete result. We discuss how we handle structures not included in the abstract language, e.g., pointers and loops, in §5.5.

Lemma 5.3: *The unfolded AFG does not contain Θ_κ-merged values and represents an equivalent constraint as the original AFG.*

Proof. (1) $\mathbb{G}_{\text{slice}}$ does not miss any Θ_{κ_i}-merged values because neither $\mathbb{G}_{\text{forward}}$ nor $\mathbb{G}_{\text{backward}}$ misses any Θ_{κ_i}-merged values. First, according to the branching rule in Figure 7, whenever a Θ_{κ_i}-merged value is defined, we have created the subgraphs, $\mathbb{G}_{\kappa_i}$ and $\mathbb{G}_{\neg\kappa_i}$. Hence, the two subgraphs can reach all uses of Θ_{κ_i}-merged values. Since the graph $\mathbb{G}_{\text{forward}}$ include all vertices reachable from $\mathbb{G}_{\kappa_i}$ and $\mathbb{G}_{\neg\kappa_i}$, $\mathbb{G}_{\text{forward}}$ does not miss any Θ_{κ_i}-merged values. Second, $\mathbb{G}_{\text{backward}}$ does not miss any Θ_{κ_i}-merged values because it is obtained by traversing the AFG from each Θ_{κ_i}-merged value.

Since $\mathbb{G}_{\text{slice}}$ does not miss any Θ_{κ_i}-merged values and each Θ_{κ_i}-merged value is reachable from $\mathbb{G}_{\kappa_i}$ and $\mathbb{G}_{\neg\kappa_i}$, all Θ_{κ_i}-merged values are replaced by either their first or second operands. Hence, the resulting AFG does not contain any Θ_{κ_i}-merged value.

(2) Recall that we copy $\mathbb{G}_{\text{slice}}$ twice, one connected to $\mathbb{G}_{\kappa_i}$ and the other connected to $\mathbb{G}_{\neg\kappa_i}$. Apparently, this copy operation does not change the number of paths in the AFG as well as the constraint represented by each AFG path.

Replacing a Θ_{κ_i}-merged value reachable from $\mathbb{G}_{\kappa_i}$ with its first operand also does not change the constraints represented by the AFG, due to two reasons. First, by Lemma 5.1, any AFG path from $\mathbb{G}_{\kappa_i}$ to a Θ_{κ_i}-merged value represents the path constraint of a program path from the true branch of $\mathbf{if}_{\kappa_i}$-statement. Second, by definition of Θ_{κ_i}, in such a program path, the Θ_{κ_i}-merged value equals its first operand. Similarly, replacing a Θ_{κ_i}-merged value reachable from $\mathbb{G}_{\neg\kappa_i}$ with its second operand does not change the constraints represented by the AFG, either. □

Lemma 5.4: *AFGs before and after decomposition are equivalent in representing path constraint.*

Proof. Vertical decomposition does not change AFG and, thus, does not change the constraint represented by AFG.

Given an AFG with multiple entry vertices, horizontal decomposition splits it into multiple subgraphs, each containing and only containing the vertices and edges reachable from one entry vertex. In other words, each path in a subgraph is a copy of the original AFG, and, for any path in the original AFG, there is a subgraph containing a copy of the path. This means the number of paths and the constraint in each path are not changed after horizontal decomposition. Hence, the constraint represented by the AFG is not changed after horizontal decomposition. □

Lemma 5.5: *If an AFG with multiple vertices cannot be vertically decomposed, each subgraph after horizontal decomposition contains a single vertex or can be vertically decomposed.*

Proof. If an AFG with multiple vertices has only one entry vertex, it at least can be vertically decomposed into two subgraphs, one is the entry vertex and the other is the remaining subgraph. Let us prove this by contradiction. If we cannot vertically decompose it to the entry vertex v and the remaining subgraph $\mathbb{G}$, it must be in the following two cases.

(1) The vertex v is not connected to all entry vertices of $\mathbb{G}$. This means that there exists an entry vertex v' in $\mathbb{G}$ that does not have any predecessors in the original graph. This further implies that the original graph has multiple entry vertices, v and v', which contradicts our assumption that the original AFG has only one entry vertex.

(2) The vertex v is connected to all entry vertices of the subgraph $\mathbb{G}$ and, meanwhile, connects to a non-entry vertex v'' in $\mathbb{G}$, which is reachable from an entry vertex v'. This means that there are two paths in the program with path constraints $v \wedge v' \wedge v''$ and $v \wedge v''$. We cannot write such a program in our abstract language (Figure 5). This is basically because, whenever we have a branching condition v', we must have the other branching condition $\neg v'$ that is connected to v.

As discussed above, an AFG that has multiple vertices but cannot be vertically decomposed must have multiple entry vertices. By definition, horizontally decomposing the graph leads to multiple subgraphs, each of which starts from a single entry vertex. As discussed before, a subgraph containing multiple vertices but a single entry vertex can be vertically decomposed. □

Lemma 5.6: *Switching the position of subgraphs in VD yields an AFG that represents an equivalent constraint as the original AFG.*

Proof. By definition, the AFG, $\cdots \bowtie \mathsf{AFG}(\rho_i) \bowtie \cdots \bowtie \mathsf{AFG}(\rho_j) \bowtie \cdots$, represents the constraint $\cdots \wedge \rho_i \wedge \cdots \wedge \rho_j \wedge \cdots$, denoted as ρ.

Switching the position of $\mathsf{AFG}(\rho_i)$ and $\mathsf{AFG}(\rho_j)$ yields the AFG, $\cdots \bowtie \mathsf{AFG}(\rho_j) \bowtie \cdots \bowtie \mathsf{AFG}(\rho_i) \bowtie \cdots$, which represents the constraint $\cdots \wedge \rho_j \wedge \cdots \wedge \rho_i \wedge \cdots$, denoted as ρ'.

By the commutative law of conjunction, ρ is equivalent to ρ'. Hence, switching the position of two subgraphs in vertical decomposition does not change the constraint represented by the AFG. □

Lemma 5.7: *Algorithm 2 yields an ordered AFG, which represents an equivalent constraint as the input AFG.*

Proof. (1) Algorithm 2 transforms an input AFG by decomposition or switching positions in vertical decomposition. By Lemma 5.4 and Lemma 5.6, these operations do not change the constraint represented by the AFG. Hence, the resulting AFG of Algorithm 2 represents an equivalent constraint as the input AFG.

(2) The resulting AFG must be ordered, which is proved below. Assume that there is a path from a vertex v_1 to a vertex v_2 in the input AFG and, the byte index in v_2 is less than that in v_1.

By Lemma 5.5, Algorithm 2 can recursively split the AFG by vertical and horizontal decomposition until every subgraph after decomposition contains a single vertex. Whenever vertical decomposition succeeds, Algorithm 2 will try to reorder the subgraphs and create the array $\mathcal{A} = [\mathbb{G}_1, \mathbb{G}_2, \ldots]$ (Lines 3-5). These subgraphs contain mutually exclusive byte indices and all byte indices in $\mathbb{G}_i$ must be less than those in $\mathbb{G}_{i+k} (k \neq 0)$.

(2.1) If v_1 and v_2 are in two different subgraphs $\mathbb{G}_i, \mathbb{G}_j \in \mathcal{A}$, it is apparent the two vertices have been correctly reordered. After reordering, the subgraphs $\mathbb{G}_i$ and $\mathbb{G}_j$ will be independently reordered. Thus, further reordering does not change the order of v_1 and v_2.

(2.2) If v_1 and v_2 are in the same subgraph $\mathbb{G}_i$, then $\mathbb{G}_i$ may be a merged subgraph (Lines 7-11) or $\mathbb{G}_i$ is obtained by vertical decomposition and, thus, cannot be vertically decomposed again (Line 13). No matter in which case, the subgraph $\mathbb{G}_i$ will be horizontally decomposed. That is to say, whenever v_1 and v_2 are not correctly ordered as (2.1), they will be in the same subgraph and the subgraph will be horizontally decomposed.

By definition, continuous horizontal decomposition will let each subgraph contains fewer and fewer paths until that the two vertices v_1 and v_2 are in a single AFG path. Given a single AFG path, it is easy to check that Algorithm 2 will correctly reorder the two vertices as (2.1).

To sum up, for any pair of vertices, v_1 and v_2, in an AFG path, Algorithm 2 will correctly order them. Hence, the resulting AFG produced by Algorithm 2 must be ordered. □