

Tupni: Automatic Reverse Engineering of Input Formats

Weidong Cui
Microsoft Research
Redmond, WA 98052
wdcui@microsoft.com

Marcus Peinado
Microsoft Corporation
Redmond, WA 98052
marcuspe@microsoft.com

Karl Chen
University of California
Berkeley, CA 94720
quarl@cs.berkeley.edu

Helen J. Wang
Microsoft Research
Redmond, WA 98052
helenw@microsoft.com

Luiz Irun-Briz
Microsoft Corporation
Redmond, WA 98052
luibriz@microsoft.com

ABSTRACT

Recent work has established the importance of automatic reverse engineering of protocol or file format specifications. However, the formats reverse engineered by previous tools have missed important information that is critical for security applications. In this paper, we present Tupni, a tool that can reverse engineer an input format with a rich set of information, including record sequences, record types, and input constraints. Tupni can generalize the format specification over multiple inputs. We have implemented a prototype of Tupni and evaluated it on 10 different formats: five file formats (WMF, BMP, JPG, PNG and TIF) and five network protocols (DNS, RPC, TFTP, HTTP and FTP). Tupni identified all record sequences in the test inputs. We also show that, by aggregating over multiple WMF files, Tupni can derive a more complete format specification for WMF. Furthermore, we demonstrate the utility of Tupni by using the rich information it provides for zero-day vulnerability signature generation, which was not possible with previous reverse engineering tools.

Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection; D.2.5 [Software Engineering]: Testing and Debugging

General Terms

Security

Keywords

protocol reverse engineering, binary analysis

1. INTRODUCTION

Recent work [6, 12, 26, 42] has established the importance of automatic reverse engineering of protocol or file format specifications. For example, the availability of such specifications gives security applications like firewalls [5,41] or intrusion detection systems [32,33] the context information of a network communication

or file parsing session, which is crucial for accurately detecting or preventing intrusions. Being able to automatically reverse engineer such protocol or file format specifications alleviates the time-consuming and error-prone manual effort.

Discoverer [12] and Polyglot [6] pilot the initial explorations in this research direction. Discoverer assumes no software access and performs automatic reverse engineering purely from network traces. Polyglot assumes software access and performs dynamic data flow analysis for reverse engineering. Both Discoverer and Polyglot reverse engineer network messages as a “flat” sequence of fields. With software access and dynamic data flow analysis, Polyglot has the advantage of being able to reverse engineer binary fields and certain dependencies (e.g., length fields). In addition to Polyglot, Lin *et al.* in [26] and Wondracek *et al.* in [42] developed new tools to reverse engineer network message formats by observing how a program processes network messages.

However, the formats reverse engineered by previous tools have missed important information that is critical for security applications. First, many input formats include arbitrary sequences of data elements (*records*). For example, most media files consist of sequences of chunks of compressed media data. Second, input fields may have arbitrary dependencies which cannot be captured by pre-defined semantics. For example, there are many different ways to compute checksums. The ShieldGen system [14] has shown that it is important to understand record sequences and various data dependencies in an attack instance for constructing a high-quality vulnerability signature for a zero-day vulnerability.

In this paper, we present Tupni, a tool that can reverse engineer an input format with a rich set of information, given one or more inputs of the unknown format and a program that can process these inputs. The main novelty in Tupni is the identification and analysis of *arbitrary* record sequences. Unlike previous tools that either ignore record sequences [6] or only work for some special cases [26,42], Tupni can identify arbitrary record sequences by analyzing loops in a program, using the fact that a program usually processes an unbounded record sequence in a loop. Tupni can also cluster records into a small set of types based on the set of instructions that process a record. In addition, Tupni can infer constraints of various, not pre-defined dependencies across fields or messages (e.g., checksums or sequence numbers) by tracking symbolic predicates from dynamic data flow analysis. Furthermore, to mitigate a fundamental problem of dynamic analysis that our view is limited by the execution path associated with a particular input, Tupni can derive a more complete format specification by aggregating the format information inferred from multiple inputs.

We have implemented a prototype of Tupni and evaluated it on ten different inputs: five file formats (WMF, BMP, JPG, PNG and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CCS'08, October 27–31, 2008, Alexandria, Virginia, USA.
Copyright 2008 ACM 978-1-59593-810-7/08/10 ...\$5.00.

TIF) and five network protocols (DNS, RPC, TFTP, HTTP and FTP). We manually compared the formats reverse engineered by Tupni with published format specifications to evaluate Tupni’s accuracy. Our experimental results show that Tupni can correctly identify all fields except for those that were ignored by the program, and it can identify all record sequences in the published format specification. We also show that, by aggregating over multiple WMF files, Tupni can derive a more complete format specification for WMF. Furthermore, we demonstrate the utility of Tupni by using its reverse engineered format for zero-day vulnerability signature generation, which was not possible with previous reverse engineering tools.

The rest of this paper is organized as follows. After defining the goals of this paper in Section 2, we describe the design of Tupni in Section 3. Then we present our evaluation results in Section 4 and demonstrate the utility of Tupni for zero-day vulnerability signature generation in Section 5. We compare Tupni to related work in Section 6 and discuss its limitations and potential future research directions in Section 7. Finally, we conclude the paper in Section 8.

2. GOALS

2.1 Scope of the Problem

Most application-level protocols involve the concept of an application *session*, which consists of a series of *messages* exchanged between two hosts that accomplishes a specific task. Correspondingly, there are two essential components in an application-level protocol specification: *protocol state machine* and *message format*. The former characterizes all possible legitimate sequences of messages, while the latter specifies the format for all possible legitimate messages. Files are a special case of protocols in the sense that each file is a single “message” and there is no “session” concept in a file specification.

In this paper, we focus on deriving the network message or file format and leave the inference of the protocol state machine to future work. We uniformly refer to both network message formats and file formats as *input formats*. We assume the boundaries of network messages can be identified.

2.2 Goals

Our goal is to design an algorithm that, given one or more inputs (files, network messages) of an unknown format and an application that can process these inputs, outputs a specification of the input format. More precisely, the format specification we seek to generate contains the following pieces of information:

- **Field boundaries:** An input instance (e.g., a particular RPC request) typically is a sequence of fields. We aim to recover the boundaries of the fields in the input.
- **Record sequences:** The identification of base fields by itself is sufficient for simple fixed input formats, i.e., formats such as TFTP in which every input instance has the same fields and only the field values differ. Most input formats do not fall into this simple class. A very common pattern is for a format to allow arbitrary sequences of records. Expressed in BNF notation, such record sequences have the form $(R_1|R_2|\dots|R_n)^*$, where R_1, R_2, \dots, R_n are different record types each of which may comprise different sequences of fields. An example of a record sequence are the property-value pairs (records) that may appear in arbitrary sequences in HTTP inputs. Other examples are records in WMF files, video and audio chunks in a number of multimedia and streaming formats (WMA, WMV, ASF, AVI, MPG,

etc.) and tags in HTML. One of our goals is to recognize such sequences.

- **Constraints:** In addition to the structural information provided by knowledge of fields and record sequences, we aim to derive information about the values of fields. In the simplest case, the input format may mandate that certain fields must have certain constant values. More generally, fields in a valid input may have to satisfy certain constraints. Examples are length fields, where one field specifies the length of an array field, and checksum fields, where the value of the checksum field depends on the values of other (possibly all other) fields in the input.

3. DESIGN

We begin by giving a high-level overview of the algorithm. Figure 1 shows the sequence of processing stages performed by Tupni. A raw input is first segmented into basic fields (Section 3.3). The next processing stage identifies record sequences (Section 3.4). We then classify the records we have identified in the previous stage (Section 3.5). This gives us record types. We look for constraints at several processing stages. Our techniques for identifying constraints are described in Section 3.6.

Most of the processing stages combine a baseline algorithm with an error correction scheme. The baseline algorithm captures the main idea underlying the processing stage. The error correction scheme accounts for the fact that real world applications do not always conform to the baseline algorithm.

Most of the descriptions in this section show how Tupni analyzes a single run of a parsing application on a particular input. At the end of the section, we describe how Tupni aggregates the results of its analysis over several runs of the same application on different inputs. For a single run of the application, the sequence of instructions that is executed during this run is called the *execution trace* of the run. Each execution trace is associated with the list of binaries that were loaded during the run and the base addresses at which the binaries were loaded. This information is readily available from tools such as iDNA [4] that capture program execution traces.

We refer to the byte positions in the input as *offsets*. We use the term *position* to identify instructions in the execution trace. For example, a particular `mov` instruction in the application binary may appear at multiple positions in the execution trace. We refer to sequences of contiguous positions in the execution trace as *subsequences*.

3.1 Background

Tupni assumes the availability of a taint tracking engine such as [10, 11, 31, 38]. These systems associate data structures with addresses in the application’s address space and update them as the application executes. In the simplest case, the data structures indicate whether the value stored at an address depends on input. When input data arrive in the application’s address space, the memory locations storing them are marked as *tainted*. Whenever an instruction reads and writes data, the data structure for the destination operand is updated depending on whether any of the source operands was tainted.

More complex data structures allow more detailed information to be tracked, including which bytes in the input the value at a tainted memory location depends on or how that value was computed. We call the latter data structure a *data flow graph* [10].

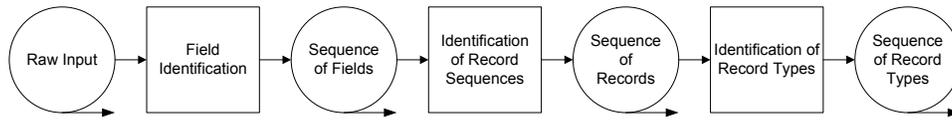


Figure 1: The processing stages of Tupni

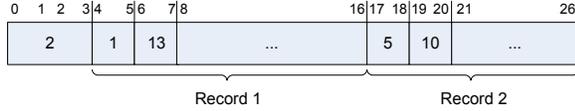


Figure 2: Running example input.

```

1 BYTE input[ BUF_SIZE ]
2 ULONG num_records, i=0, offset=4
3 USHORT record_type, record_size
4 FCT_PTR record_hdlr[NUM_REC_TYPES]

5     call read_input( input )
6     mov  num_records, (ULONG *)&input[0]

7 loop: cmp  i, num_records
8       jeq  end

9     mov  record_type, (USHORT *)&input[offset]
10    mov  record_size, (USHORT *)&input[offset+2]

11    call record_hdlr[record_type](&input[offset])

12    add  offset, record_size
13    inc  i
14    jmp  loop
15 end:

```

Figure 3: Example application.

3.2 Running Example

Throughout this section we will use a running example to help illustrate the stages of Tupni. For ease of exposition, we use a very simple synthetic input format and a similarly simple application that processes inputs from this format.

The input format consists of a header followed by a sequence of records. The header has only one field. It specifies the number of records that follow. Each record starts with a two-byte field that specifies the record type, followed by a second two-byte field that specifies the size of the record in bytes. The remaining bytes of each record depend on the record type. This simple format is structurally similar to real world formats such as WMF, PNG, WMA and WMV.

Figure 2 shows an example input from this format. The first field specifies the number of records as 2. The next 13 bytes make up the first record, and the following ten bytes make up the second record. These lengths are specified in the second field of each record. The first field of each record specifies its type (as types 1 and 5, respectively).

Figure 3 shows a simple application that parses inputs from our example format. Our pseudo assembly notation can be mapped to working x86 machine code by assigning x86 registers to our variables and adding instructions to pass function call parameters. Line 5 reads the input into a sufficiently large buffer (`input`). Line 6 reads the first four bytes of `input` into `num_records`, which tells the application how many records it needs to process. Each iteration of the following loop (lines 7 to 14) processes one record. Lines 7 and 8 exit the loop after the last record has been

processed (`i == num_records`). Lines 9 and 10 read the type field and the length field of the record. Line 11 calls the handler function for the given record type. We assume that the function pointer array `record_hdlr` has been initialized with a handler function for each record type. These handler functions are part of the application, but not displayed in Figure 3 to keep the example manageable. Finally, line 12 sets `offset` to the beginning of the next record, and line 13 increments the index variable that counts how many records have been processed.

The execution trace of running this program on the example input is `5, 6, 7, . . . , 14, 7, . . . , 14, 7, 8, 15`. That is, after executing lines 5 and 6, the trace contains two full iterations of the loop. Each segment `7, . . . , 14` contains the instructions of the handler function between lines 11 and 12.

3.3 Field Identification

This first step partitions the input into short sequences of consecutive bytes that are likely to correspond to basic fields in the input format. Typically, several consecutive input bytes will be mapped to a larger field such as a 32-bit integer. We deduce this mapping by observing the operands of CPU instructions. Given an operand of a CPU instruction in the execution trace, we call the longest sequence of tainted bytes from contiguous offsets contained in the operand a *chunk*. Our algorithm identifies all chunks in the execution trace by stepping through the trace and inspecting the operands of each instruction for taint. We ignore move instructions as they do not process the operand and contain little information about field sizes.

Any byte of the input that is read by the application will be contained in at least one chunk. In general, a chunk may be accessed by multiple instructions. We keep track of this by assigning a weight $w(c)$ to each chunk c . The weight of a chunk is simply the number of instructions in the execution trace that access the chunk (ignoring move instructions).

In this way, by observing the operands of x86 instructions, we can identify chunks contained in 8-, 16-, 32- and 64-bit integer operands as well as floating point chunks. We can also recognize big-endian integers if we see a chunk with reversed bytes. Currently we track input chunks at byte granularity; we believe it takes only engineering efforts to refine this to bit granularity.

Running Example.

The first time the application reaches line 7, the second operand of the `cmp` instruction is tainted with the first four bytes of the input. This causes Tupni to create a chunk `[0, 3]` for offsets 0 to 3 and to assign it weight 1. Note that the `mov` instruction in line 6 does not create a chunk. It simply propagates taint into `num_records`. Analogous statements apply to lines 9 and 10.

The next chunk `[4, 5]` is created in line 11 because `record_type` is tainted with offsets 4 and 5. The instructions executed by the record handler function called in line 11 will typically create further chunks for the offsets inside the record. Since our simple example does not specify handler functions, we ignore this part of the execution trace. Line 12 creates the chunk `[6, 7]` since `record_size` is tainted with offsets 6 and 7.

In the second loop iteration, line 7 increases the weight of $[0, 3]$ to 2, and lines 11 and 12 act as in the previous iteration, creating chunks $[17, 18]$ and $[19, 20]$, respectively. Finally, the third execution of line 7 raises the weight of $[0, 3]$ to 3. The final set of chunks is $[0, 3]$, $[4, 5]$, $[6, 7]$, $[17, 18]$ and $[19, 20]$; plus any chunks generated by the handler functions.

3.3.1 Error Correction

The next step is to map chunks to fields. A general difficulty is that, during the course of its execution, an application may access the same offset using different operand types. Thus, an offset may be contained in different chunks. The most common application behavior that gives rise to such inconsistencies includes (a) bulk accesses (memory copies, checksums, etc.) in which the application reads a part of the input without concern for field boundaries and (b) optimized string processing code that uses 32-bit (or larger) operations to process an ASCII or unicode string.

The goal is to find a consistent subset S of chunks such that we can identify each chunk in S with a field. Tupni has to find a subset S of chunks such that all chunks in S are disjoint and the combined weight of the chunks in S is maximized. This is an instance of the weighted Maximum k -Set Packing problem. While it is intractable to even approximate the optimal solution under worst-case analysis [21], the problem instances we obtain from real-world execution traces are quite simple with little overlap among chunks. We use the greedy algorithm [7] to compute S and identify the chunks in S with input fields. Based on our evaluation (Section 4), this simple algorithm seems sufficient.

In general, the fields we have found may not cover the entire input. In many cases, the application never accesses certain parts of the input. Thus, observing the application reveals no information about these parts. All Tupni can do is to mark the contiguous gaps between fields as virtual fields.

In our simple example, all chunks are disjoint. Thus, Tupni promotes all chunks into fields. Again, ignoring the handler functions, offsets $[8, 16]$ and $[21, 26]$ are gaps between fields. Tupni adds $[8, 16]$, $[21, 26]$ as virtual fields.

3.3.2 Strings

We consider strings as regular record sequences. Thus, we have no string fields. Given a string, Tupni will recognize the string characters as fields. The next section will describe how Tupni infers that these character fields form a string as part of the more general framework for recognizing record sequences.

A special problem arises for strings because many applications use optimized string processing code that uses 32-bit rather than 8-bit or 16-bit instructions to access strings. This could cause string characters to be misclassified as 32-bit integers. Fortunately, the optimized string processing code is easy to recognize. All instances we have observed on Windows and Linux involve boolean and comparison operations with two or three specific constants. If we recognize optimized string processing code, we ignore the corresponding instructions in the calculation of chunk weights.

3.4 Identification of Record Sequences

This section describes how we identify record sequences. As defined in Section 2.2, a record sequence covers a contiguous sequence of offsets in the input. Our algorithm is based on the observation that applications have to use a loop or recursive calls to process an unbounded sequence of records. We have developed Tupni for the common case of loops. We believe that similar techniques can be used to handle recursive calls.

The algorithm proceeds in the following three steps: (a) Identify

all loops that are executed in the execution trace; (b) For each loop, find the relevant input fields that are accessed in each loop iteration; (c) For each loop that processes relevant input fields, identify the record boundaries. The rest of the section describes each of the three steps in detail.

3.4.1 Loop Identification

The first problem is to find loops in the application. The solution is to search for cycles in the control flow graph (e.g., [3]). This method yields a complete list of loops and their relationship. We only consider loops with a single entry point.

The next task is to map the loop information to the execution trace. Given a loop (a cycle in the control flow graph), we can map each instruction in that cycle to the program counter (EIP) value at which the instruction was stored when the execution trace was created. This makes use of the information about loaded binaries mentioned in Section 3.1. For each loop, we maintain a compact representation of the set of EIP values of the instructions in the loop. We also store the EIP of the unique entry point of each loop.

Tupni tests for each position in the trace and for each loop whether the EIP value at that position is contained in the set of EIP values of the loop. The result of this process is the list of all subsequences of the execution trace that correspond to the execution of a loop (labeled with an identifier for that loop).

Finally, we identify the individual loop iterations within each subsequence. We mark the beginning of an iteration at the points where the execution trace hits the (unique) entry point of the loop.

Running Example.

The code in Figure 3 has a single loop (cycle in the control flow graph) in lines 7 to 14. Line 7 is the only entry point of this loop. Let A be a label for this loop. There may be further loops in the handler functions. Those would be recognized as nested within loop A . Since the execution trace in Section 3.2 uses line numbers rather than EIP values, there is no mapping to be done. Otherwise, we would have to look up at what base address the application binary was loaded when the execution trace was created. Comparing the line numbers in the execution trace in Section 3.2 with the line numbers of our loop (7 to 14), we find that the subsequence that starts at the third instruction in the trace and ends right before the last instruction corresponds to the execution of loop A .

Tupni identifies iterations by looking for the entry point of loop A (line 7) within the subsequence. It finds three occurrences of line 7 in the subsequence (at the very beginning, toward the middle and at the end. Thus, Tupni splits the subsequence into three iterations at these three positions. The third iteration contains only lines 7 and 8.

3.4.2 Identifying Relevant Fields

In the following, we will also use the term *loop* as a shorthand for “a subsequence of the execution trace labeled by a loop identifier”. Most loops executed by applications do not process record sequences from the input. We are only interested in those loops that iterate over multiple input fields. In this step, we identify such loops.

Consider a loop l and let n be the number of iterations that were executed. We define sets I_i ($1 \leq i \leq n$) as follows: An instruction (identified by its instruction pointer (EIP) value) is contained in I_i if the instruction accesses a field during the i -th loop iteration that it does not access in any other iteration of l .

If I_n is empty then we treat the loop as if it had only $n - 1$ iterations. This special case is necessary since the last iteration of a loop is often not a true iteration, but only a check of the termination

```

FindRecordBoundaries(
1. IN  $n$ , // number of loop iterations
2. IN  $(I_1, \dots, I_n)$ , // iteration dependent instructions for each loop iteration
3. OUT  $(s_1, \dots, s_n)$ , // start offset of each record
4. OUT  $(e_1, \dots, e_n)$  // end offset of each record
5.
6. for ( $j$  from 1 to  $n$ )  $s_j = -1$ ; // mark start of record  $j$  as unknown
7.
8. for ( $j$  from 1 to  $n$ ) {
9.   if ( $s_j == -1$ )
10.    then  $s_j = \min\{\text{Offset}(\text{Field}(\text{inst})): \text{inst} \in I_j\}$ 
11.     $I = \{\text{inst} \in I_j : \text{Offset}(\text{Field}(\text{inst})) = s_j\}$ 
12.    for ( $i$  from  $j + 1$  to  $n$ )
13.      if ( $I \cap I_i \neq \{\}$ )
14.         $s_i = \min\{\text{Offset}(\text{Field}(\text{inst})): \text{inst} \in I_i \cap I\}$ 
15. }
16.
17. for ( $j$  from 1 to  $n - 1$ )  $e_j = s_{j+1} - 1$ ;
18.  $e_n = \max\{\text{Offset}(\text{Field}(\text{inst})) + \text{sizeof}(\text{Field}(\text{inst})): \text{inst} \in I_n\}$ 

```

Figure 4: Identification of record boundaries

condition (as in our example, where the third iterations consists only of lines 7 and 8).

We call $\bigcup_i I_i$ the set of *iteration dependent* instructions. We call a loop *iteration dependent* if $I_i \neq \{\}$ for all $i \in \{1, \dots, n - 1\}$.

We compute the sets I_i for all loops identified in the previous step. At the end of this analysis, we obtain the list of iteration dependent loops in the execution trace and, for each iteration dependent loop, the sets I_i of its iteration dependent instructions.

Running Example.

$I_1 = \{9, 10, 11\}$ and $I_2 = \{9, 10, 11\}$. Thus, the loop is iteration dependent. For example, line 9 accesses field $[4, 5]$ in the first iteration and only in the first iteration. Thus, it belongs to I_1 , It also accesses field $[17, 18]$ in the second iteration and in no other iteration. Thus, it also belongs to I_2 . On the other hand, line 7 is not iteration dependent. It accesses the same field $([0, 3])$ in all iterations.

3.4.3 Identifying Record Boundaries

In this step, we group fields into records and identify the record boundaries. We make use of the fact that, given the definitions of Section 2.2, a record is a contiguous sequence of fields and a record sequence consists of contiguous records. Furthermore, we assume that the loop processes records in the order in which they appear in the input. Loops that do not satisfy this assumption are ignored. However, we can handle loops whose iterations access fields in records outside the record currently being processed.

Our algorithm is shown in Figure 4. Tupni calls this algorithm for every iteration dependent loop. For ease of presentation, the figure assumes that each iteration dependent instruction `inst` accesses only one field (`Field(inst)`). This assumption has no fundamental importance. Its only purpose is to remove tedious details from Figure 4.

For any field f , let $\text{Offset}(f)$ denote the offset in the input of the first byte of f . The algorithm sets the start of the first record to $\min\{\text{Offset}(f)\}$, where the minimum is taken over all fields that are accessed by iteration dependent instructions in the first iteration of the loop (line 10). Next, in line 11, it identifies the set I of iteration dependent instructions that access the field at the beginning of the first record in the first iteration. The scheme now uses the heuristic that the instructions in I are likely to access the beginning of a record whenever they appear. Thus, it looks for instructions from I in each iteration (line 13) and sets the start of the corresponding record accordingly (line 14). The process is re-

peated (line 8) for the case that an iteration does not use any instruction from I . Finally, the end of each record is set to the offset immediately before the start of the next record (line 17). The end of the last record is set to the last input byte accessed by any iteration dependent instruction in the last iteration (line 18). This procedure is performed for all iteration dependent loops. A loop is ignored if the computed record start addresses are not increasing.

Running Example.

The algorithm sets $s_1 = 4$ (line 10), since $I_1 = \{9, 10, 11\}$, $\text{Field}(9) = [4, 5]$ and $\text{Offset}([4, 5]) = 4$. That is, 4 is the smallest offset accessed by instructions in I_1 . Line 11 sets $I = \{9, 11\}$ since both instructions accessed field $[4, 5]$ in the first iteration. Next, it sets $s_2 = 17$ since instructions 9 and 11 access field $[17, 18]$ in the second iteration. Line 17 sets $e_1 = s_2 - 1 = 16$. Finally, line 18 sets $e_2 = 20$ the largest offset accessed by an instruction from I_2 . While all other values are correct, e_2 is somewhat too small. This is primarily the result of the fact that we have ignored the handler functions called in line 11 of the example application.

3.4.4 Length Determination

We also output for each record sequence how its length is determined. We consider the following three cases: (a) The length is determined by a termination record (e.g., null terminated ASCII strings); (b) The length is determined by a separate length field; (c) The length is fixed and implicitly determined by the protocol specification.

Tupni identifies case (a) (termination record) by checking all equality comparisons between a constant and the value of a field accessed by an iteration dependent instruction. We classify the record sequence as case (a) if we observe a constant for which there is such an equality check that evaluates to false in each iteration (up to but not including the last iteration) and if the last iteration contains an equality check with the same constant that succeeds. For case (b) (length fields), we use a technique from Polyglot [6]: We identify fields that are accessed to compute the termination condition of loops. If we do not recognize case (a) or (b), we classify the record sequence as case (c) (implicit length).

3.5 Record Type Identification

Typical formats specify record sequences such that records come from a small set of classes (the record types of Section 2.2). So far, we have identified record sequences, found out how the length of the record sequence is determined and found the boundaries of the individual records in a sequence. In this step, Tupni tries to find the record type for each record. More precisely, Tupni considers two records as belonging to the same record type if the loop iterations that process the records execute “mostly” the same instructions. This is based on the observation that applications typically have a separate handler function for each record type.

The problem is complicated by the fact that record sequences may be nested. That is, there are sequences whose records contain other sequences. If a record contains a child record sequence then the instructions that process it may depend heavily on the particular instance of the child record sequence it contains. Thus simply looking for similar instruction sequences will not solve the problem.

Let l be an iteration dependent loop, and let r be the record sequence l processes. Let n be the number of records in r . For $i \in \{1, \dots, n\}$, we define Q_i to be the subset of instructions from I_i that access fields in the i -th record of r . We compute the sets Q_i for every record sequence we have identified in the previous step.

The set Q_i is a superset of the set of instructions that we consider

relevant. As outlined above, if records in r contain nested record sequences then records of the same record type may have widely different sets Q_i . Therefore the next step of our algorithm is to transform each set Q_i into a set Q'_i as follows.

Tupni collapses every segment of the instruction trace that contains an execution of a child loop into a single virtual instruction. This instruction is assigned a virtual identifier (virtual EIP) such that two virtual instructions have the same identifier if and only if they correspond to executions of the same loop. Q'_i is set to Q_i , except that all instructions in the child loop execution are removed from the Q_i and replaced by the identifier of the virtual instruction that represents the child loop. The net effect of this transformation is that iteration dependent loops that process child record sequences contribute a single virtual instruction to the Q'_i of the parent record. This way, it only matters whether the child loop was executed, but not what happened during its execution.

In summary, Tupni considers each record sequence r that was identified in the previous step and computes Q'_i for every record i in r . Tupni classifies two records i, j as having the same type if $Q'_i = Q'_j$.

Running Example.

Recall that, in the previous step, Tupni had identified one record sequence with two records. We have $Q_1 = \{9, 10, 11\} \cup h_1$ and $Q_2 = \{9, 10, 11\} \cup h_5$, where h_1 and h_5 denote the instructions in `record_hdlr[1]` and `record_hdlr[5]`, respectively, that access the record sequence. Typically, h_1 and h_5 will be different.

If neither of the two records contains a nested record sequence then $Q'_1 = Q_1$ and $Q'_2 = Q_2$. In order to examine the other case, let `record_hdlr[1]` contain a loop that iterates byte-by-byte over the payload of record 1 (offsets 8 to 16). In this case, h_1 contains up to 9 different instructions (for the nine loop iterations from offset 8 to offset 16). Q'_1 would now be $Q'_1 = \{9, 10, 11, v_1\}$, where v_1 is a unique identifier for the loop in `record_hdlr[1]`.

3.6 Constraint Identification

This section describes how Tupni identifies conditions on the values of fields that an input must satisfy to be valid. This includes relations between the values of multiple fields. We call such conditions *constraints*. Tupni considers three types of constraints.

Symbolic Predicates.

The dynamic data flow analysis engine [10] used by Tupni outputs a sequence of conditions on the input (symbolic predicates) that were checked by the application during its execution. These conditions encode the application’s execution path. In general, some of these conditions will represent properties that any valid input must have while other conditions simply represent properties of the particular input the application was run on. For the purpose of gathering constraints on the format, we are interested in the former, but not in the latter.

Tupni generates constraints on the value of single fields (single-value constraints) as follows. For each input field i , Tupni outputs the conjunction of all symbolic predicates of the form $f(\text{input}[i]) \{<, =, >, \neq\} y$, where f is a function that depends on nothing but input values and constants that are hard coded into the application (e.g., immediate operands of x86 instructions). In addition, we ignore all symbolic predicates associated with checks for termination values of record sequences, as the information contained in them is already accounted for by the analysis of length determination (Section 3.4).

Tupni also outputs symbolic predicates of the form `input[x] =`

$f(\text{input}[y], \text{input}[z], \dots)$, where f is a function that depends on nothing but input values and constants that are hard coded into the application. This type of constraint can capture many types of checksum calculations. We call such constraints *functional constraints*.

The constraints generated by this simple procedure may be particular to the input the application was run on and may not represent the input format in general. This is a general limitation of dynamic analysis. Recent systems such as Bouncer [9] and ShieldGen [14] were designed to overcome this limitation in a different application domain. We believe that the techniques employed by these systems could be used to eliminate most of the remaining input specific conditions.

Inter-Message Dependencies.

An inter-message constraint for a multi-message network protocol determines the value of a field in a later message based on the values of one or more fields in an earlier message. Typical examples include session IDs and sequence numbers.

Tupni inspects all attempts by the application to send data to the network. If it finds tainted data in an outgoing message, it outputs the associated data flow graph that describes how the tainted output field depends on the previous input.

Length Fields.

Tupni uses three techniques for identifying length fields. The first technique makes use of the semantics of certain platform functions as described in Section 3.7. The other two techniques are similar to techniques used in Polyglot: identifying fields used to compute pointers that are used to access other input fields and identifying fields that are accessed to compute the termination condition of loops that process record sequences.

3.7 Platform-Specific Functions

Knowledge of the semantics of platform specific functions such as system calls, application programming interface (API) functions and functions in runtime libraries can significantly enhance the accuracy and functionality of Tupni. The idea of using the semantics of library calls has been used in Bouncer [9]. However, the way in which Tupni makes use of function calls is quite different from the method in Bouncer. As a preliminary step, we have added specifications of six string processing functions and 20 memory allocation functions to Tupni. These specifications are used in the following areas:

- Field identification: Use of an input chunk as a function parameter can provide strong evidence that the chunk is a field. This evidence is incorporated into the field recognition step by increasing the weight $w(c)$ of the chunk if Tupni sees a call to a known function in the execution trace. More extensive use of function semantics has the potential of providing very detailed information about the types and semantics of input fields. We leave this for future work.
- Identification of record sequences: Functions may have record sequences (or pointers to them) as parameters. For example, a call to a standard string processing function with a `char*` pointer that points to tainted data provides strong evidence that the pointer marks the start of a string.
- Constraint identification: function calls may provide information about different kinds of constraints. We have used knowledge of memory allocation functions to identify length fields. For example, a call to `malloc` with a tainted size

parameter provides evidence that this parameter is a length field.

In general, we do not consider the parts of the execution trace that are spent inside any of the known functions in our analysis. Effectively, this collapses calls to these functions into a single virtual instruction with special semantics.

3.8 Output Format

```

Input ::= F1 :D
      S1[F1]

S1    ::= R1 |
      R2 |
      ...

R1    ::= F2 :W
      F3 :W { F3 = SIZE(R1) }
      ...

R2    ::= F4 :W
      F5 :W { F5 = SIZE(R2) }
      ...

```

Figure 5: The output format for the running example.

At this point, Tupni’s analysis of a single input is complete. Tupni outputs the fields and record sequences it identifies in an enhanced BNF format. An example of this output format is shown in Figure 3.8 for the running example. Tupni generates a rule that lists the top-level fields and record sequences in the order of their positions in the input. We refer to this rule as the *root rule*. For each record sequence, Tupni generates an alternation rule to cover all its record types. For each record type Tupni generates a rule to list its fields and child record sequences. This process continues recursively for all record sequences and record types. In addition, Tupni outputs constraints in the enhanced BNF format.

3.9 Aggregation over Multiple Inputs

So far, we have discussed how Tupni analyzes a single execution trace – corresponding to a single input. If multiple inputs of the unknown format are available, the application can be run on each of them, and Tupni can perform its analysis on each of the resulting execution traces. The last step is to combine the individual results into a single format specification.

To do so, Tupni matches fields, record sequences, and record types across different execution traces. Consider two execution traces from the same application, but different inputs. Tupni considers two base fields from two execution traces to match if they are accessed by the same set of instructions. Tupni considers two record sequences from two execution traces to match if they are processed by the same loops (identified by the unique loop entry point). Finally, Tupni considers two records from matching record sequences to have the same type if their respective sets Q'_i are equal.

After identifying all matching fields, record sequences and record types in the two execution traces, Tupni merges the BNF rules for each of the execution traces into a single set of BNF rules. Tupni first identifies the pairs of BNF rules that can be merged. These include the root rules and the BNF rules of matched record sequences and record types. To merge BNF rules of matched record sequences, Tupni simply creates a new alternation rule that includes all record types in the old rules. Tupni merges the root rules and rules of matched record types in two steps. It first aligns the list of fields and record sequences using the type-based sequence alignment technique proposed in [12]. If all fields and record sequences

are perfectly matched, it means that the two rules are identical; Tupni simply keeps one. Otherwise, it creates a new alternation rule for each pair of unmatched fields and record sequences. Then it creates a new rule that lists matched fields and record sequences as well as the alternation rules for unmatched fields or record sequences.

4. IMPLEMENTATION AND EVALUATION

We have developed a prototype of Tupni. Our dynamic data flow engine was built on a re-implementation of the Vigilante system [10]. Our prototype system uses iDNA [4], a binary program translator, to capture and replay program execution traces. Our system works on x86 instructions directly. Excluding the code of the original Vigilante and iDNA, our Tupni prototype has 14,000 lines of C++ and 4,100 lines of Perl.

In this section, we first describe our experimental setup and evaluation methodology. After that, we present our experimental results.

4.1 Experimental Setup

Our evaluation is divided into two parts. We first tested how accurately Tupni can reverse engineer the format of a single input message or file. Then we evaluated its capability of generalizing the input format over multiple inputs.

In the first set of experiments, we evaluated Tupni on both files and network messages. We tested five binary files (WMF, BMP, JPG, PNG and TIF), three binary network messages (DNS response, TFTP data and RPC bind request), and two text network messages (HTTP GET request and FTP port command). We selected these input formats because they are representative protocols and were studied in previous work. We intentionally avoided types of input formats for which we knew a priori that our prototype would perform poorly (see Section 7 for a discussion of these formats). For instance, our prototype does not work well if the field boundaries of a format do not coincide with byte boundaries due to its byte-based taint tracking. For the compressed image files among our test cases, we used Tupni to reverse engineer the container file format and ignored the format of the compressed data blobs. We used WMF, the DNS response message, and the HTTP request message to guide our design. We ran the remaining seven test cases without changing the prototype.

For each input, we recorded an execution trace of a binary program parsing and processing it on Windows XP Professional SP2. Then we fed the execution trace to Tupni to reverse engineer the format. The inputs and binary programs are listed in Table 1. In all our test cases, Tupni can reverse engineer the formats in at most 5 minutes on a 3GHz machine running Windows XP Professional SP2.

4.2 Evaluation Methodology

In our first set of experiments on single test inputs, we focused our evaluation on *accuracy*, that is, how accurately Tupni can identify the fields, record sequences, record types, and constraints in the test inputs. Tupni *automatically* outputs the reverse engineered formats in an enhanced BNF format (referred to as the *Tupni format*). In order to evaluate the accuracy of Tupni’s output, we compare the format specification Tupni produced for each test case with the published specification [1, 2, 16, 20, 28, 34, 37, 39, 40] (referred to as the *published format*) for the same format and identify discrepancies. In our second set of experiments, we focus our evaluation on *completeness*, that is, how well Tupni can infer a more complete input format over multiple inputs.

It is hard to compare our results with previous work [6, 12, 26,

Format Name	Application	Execution Trace (instructions)	Input Size (B)	Run Time (min)
WMF	gdi32.dll	3.2M	4594	5
BMP	mspaint.exe	12.0M	3126	1
JPG	gdiplus.dll	15.7M	3224	3
PNG	gdi32.dll	12.2M	3543	3
TIF	gdiplus.dll	12.0M	4337	3
DNS response	nslookup.exe	4.7M	46	<1
RPC bind	rpcss.dll	128K	164	<1
TFTP data	tftp.exe	5.5M	28	<1
HTTP request	inetinfo.exe	156M	107	3
FTP port	ftpsvc2.dll	121M	28	3

Table 1: Summary of test inputs in the evaluation.

Input	WMF	BMP	JPG	PNG	TIF	DNS	RPC	TFTP	HTTP	FTP
Error Fields	2	3	0	889	2	4	6	0	0	0
Total Fields	2257	3087	3211	1551	4220	32	137	26	23	4

Table 2: Summary of field identification results.

42] quantitatively as these tools are not publicly available, and we used different file/protocol formats for evaluation. We will give a qualitative comparison in Section 6.

In the rest of this section, we will first summarize our experimental results on the ten test cases and discuss why Tupni made errors in reverse engineering the formats. After that, we will analyze the aggregation of format information across multiple inputs.

4.3 Experimental Results on Single Inputs

4.3.1 Field Identification

To evaluate the accuracy of field identification, we counted the number of fields in the published format that were identified incorrectly by Tupni. The results are shown in Table 2.

By analyzing the execution traces and source code, we found that most of these errors have the same cause—a program ignores certain parts of an input. When this happens, Tupni will mark these gaps as virtual fields. For example, there are 127 virtual fields in the PNG test. This happens because the first 128 chunks in the PNG test file are identical, and the parsing program ignored the data bytes in all but the first chunk. Since the application never accessed the data bytes, Tupni had to create one virtual field for each chunk that covers all the data bytes in the chunk. The published specification defines seven fields on these data bytes. This results in a misclassification of 889 fields (7 fields per chunk times 127 chunks).

One of the errors in the DNS test case occurs because the program processed each byte in a two-byte flag field separately.

4.3.2 Record Sequence Identification

We evaluated the accuracy of record sequence identification by measuring the number of record sequences that are in the published format but missed by Tupni and the number of record sequences that are not in the published format but identified by Tupni. We refer to the former as *false negative* (FN) sequences and to the latter as *false positive* (FP) sequences. The results are shown in Table 3. Tupni did not miss any record sequences in the published format of any of the ten test cases. This confirms our observation that a parsing program usually parses and processes record sequences in loops.

Tupni output false positive sequences in three test cases. By analyzing the applications, we found that, in most cases, this happens

because the application uses a loop to process input fields that do not belong to a record sequence in the published format. In the PNG case, Tupni identified two false positive sequences from execution loops in decoding the image file. In the JPG test, Tupni identified the first two bytes `ffd8` as a sequence because the parsing program compared them with a two byte string to look for the start of image. Tupni found extra record sequences in the HTTP test because the program processed some substrings in loops.

4.3.3 Record Type Identification

Ideally, the set of record types identified by Tupni (referred to as the *Tupni record types*) should correspond one-to-one to the record types in the published format (referred to as the *published record types*). There may be two kinds of inaccuracies. First, records of a single Tupni record type may belong to multiple published record types. Second, records of multiple Tupni record types may belong to a single published record type. In our evaluation, we analyzed these two kinds of errors for each record type seen in the test cases. For the first kind of error, we counted the number of Tupni record types that have records belonging to multiple published record types. We will refer to them as *under-classified record types*. For the second kind of error, we counted the number of published record types that contain records assigned to multiple Tupni record types. We will refer to them as *over-classified record types*. We did not analyze the FTP and HTTP test cases in this evaluation because all their record sequences are ASCII strings and have trivial record types. The results are shown in Table 4. Tupni did not make any errors of the first kind, and it made two errors of the second kind.

By analyzing the execution traces and source code, we found that these two errors arise because a record type defined in the published specification may be an abstraction of a complex object that has multiple semantics, which causes the parsing program to behave differently on records of the same record type. In the WMF test case, the published record type associated with the error is `CreateBrushIndirect` whose parameters contain three flags that control many different behaviors of Brush objects. In the JPG test case, Tupni assigned `DHT` marker segments (records) to two different record types. We found that this happens because `DHT` marker segments can define two different types of Huffman tables (lossy and lossless) that are processed differently.

Input	WMF	BMP	JPG	PNG	TIF	DNS	RPC	TFTP	HTTP	FTP
FN Sequences	0	0	0	0	0	0	0	0	0	0
FP Sequences	0	0	1	2	0	0	0	0	3	0
True Sequences	35	0	15	4	1	4	6	1	12	3

Table 3: Summary of record sequence identification results.

Input	WMF	BMP	JPG	PNG	TIF	DNS	RPC	TFTP
Under-Classified Record Types	0	0	0	0	0	0	0	0
Over-Classified Record Types	1	0	1	0	0	0	0	0

Table 4: Summary of record type identification results.

4.3.4 Constraint Identification

In this section, we discuss Tupni’s accuracy on constraint identification. Tupni can identify three kinds of constraints: symbolic-predicate, inter-message, and length. Since length field identification has been widely studied in previous work and Tupni achieved similar performance, we will not discuss length constraints but focus on the other two kinds.

Symbolic-Predicate Constraints: Our evaluation on functional constraints was focused on whether Tupni can identify all checksum dependencies. In the ten test cases, only two input formats, WMF and PNG, have checksum fields. Tupni identified one functional constraint in the WMF test that captures the dependency between the *checksum* field in the WMF header and the rest of the header. A PNG file [40] has a sequence of chunks and each chunk has a 4-byte CRC checksum field calculated on the preceding bytes in the chunk. Tupni did not find functional constraints that reflect these checksum dependencies. By analyzing the execution trace, we found that this happens because the parsing program *ignored* the checksums while parsing each chunk.

Tupni also identified single-value constraints in every test case, many of which revealed interesting insights into the input formats. For example, Tupni inferred that the first byte of a marker segment in the JPG file must be 0xff; the size field of a draw record in the WMF file must be at least 3; there are at most 20 context elements in the RPC bind message. We also observed that there are constraints that are particular to the input. For example, Tupni inferred that the version of the WMF file must be 0x0300; however, by analyzing the source code we found that it can also be 0x0100. This is a general limitation of dynamic analysis. We will discuss this issue further in Section 7.

Inter-Message Constraints: In the ten test cases, only three network protocols, DNS, RPC and TFTP, have inter-message constraints. To identify the inter-message constraints in DNS and RPC, one must track the server program but we tracked the client program in our tests. Thus, TFTP is our only test case with an inter-message constraint. Tupni correctly identified this constraint. In particular, by tracking how the ACK message was generated after receiving the data message, Tupni identified that the AckNum in the acknowledge message must equal the BlockNum in the data message.

4.4 Experimental Results on Multiple Inputs

In this section, we present the experimental results of running Tupni over 150 WMF files to generate a more complete WMF format specification. The 150 files were randomly selected from a test template repository at Microsoft. The average file size is 1,577 bytes. The average number of draw records per file was 63. We first examine how accurately Tupni can match fields and record sequences across multiple inputs, then describe how much improvement on format coverage Tupni can achieve over multiple inputs.

We do not present results on matching record types since we have evaluated how accurately Tupni can identify record types in Section 4.3.3. To evaluate the accuracy on matching fields, we inspected the first 17 fields identified in the 150 Tupni formats generated on each individual file. These 17 fields correspond to the metafile header in a WMF file. We say two fields have the same *type* if they match. For each field position, we counted the number of different types. We found that 12 out of the 17 field positions have only one field type, which ensured a correct alignment of BNF rules from different inputs. Also, there are at most three different types for a field position. By analyzing the traces and source code, we found that the different types appear because the program took different execution paths based on different parameters (e.g., whether the file size is the same as specified in the MetaFile_Size field). There are ten different record sequences in the 150 WMF files. Tupni matched all record sequences correctly.

To evaluate the improvement on format coverage, we counted the number of distinct draw record types in each WMF file and in the 150 files together. The minimum number of distinct draw record types in a single file is 9. However, there are 80 distinct draw record types in the 150 files. The aggregated output of Tupni included these 80 record types. These results show that Tupni can significantly improve the format coverage by aggregating multiple inputs.

5. TUPNI FOR ZERO-DAY VULNERABILITY SIGNATURE GENERATION

We have shown that Tupni can automatically reverse engineer the input formats with high accuracy in the ten test cases. In this section, we evaluate Tupni’s effectiveness by using its reverse engineered formats in a real-world security application: zero-day vulnerability signature generation. Tupni can also be used for other security applications such as fuzzing tests. Fuzzing tests can tolerate more inaccuracies than vulnerability signature generation since the latter constructs a vulnerability signature based on the reverse engineered format while the former only uses it to generate new test inputs.

ShieldGen [14] is a system for automatically generating a signature for an unknown vulnerability, given a zero-day attack instance. ShieldGen requires a specification of the data format to generate new potential attack instances (probes). The architecture of the original ShieldGen system is shown in Figure 6. When presented with a potential exploit, ShieldGen feeds the suspicious traffic into an attack detector such as [10]. If the detector recognizes an attack, ShieldGen feeds the exploit into its data analyzer component [5]. The data analyzer uses the format specification to parse the input and feeds the result into the probe generator. The probe generator will generate variants of the original input (probes) and use the detector to determine if a probe still exploits the vulnerability. The

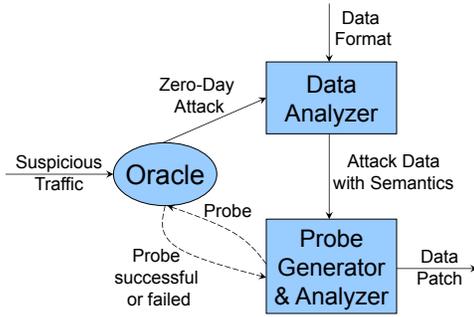


Figure 6: Original ShieldGen

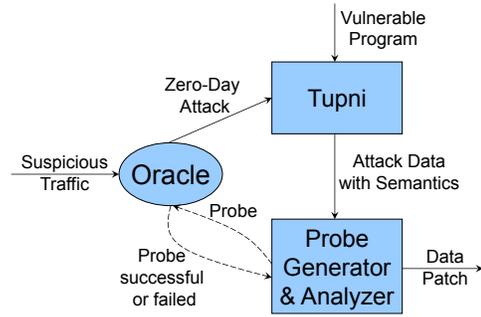


Figure 7: ShieldGen + Tupni

most important aspect of this search is the removal of all records from probes that are not required to exploit the vulnerability. Thus, information about record sequences and associated constraints in the input specification is critical for ShieldGen.

In our experiments, we replaced the Data Analyzer module with Tupni and replaced the Data Format with the vulnerable program. The new architecture is shown in Figure 7. Then we repeated the case study of the WMF vulnerability in the ShieldGen paper [14] with this new signature generation system. The new system automatically generated the *same* vulnerability signature as in [14].

None of the existing reverse engineering tools [6,12,26,42] lends itself to the ShieldGen application because they fail to reliably identify record sequences in a sufficiently broad class of input formats. In the case of WMF, without identifying record sequences, the system will not be able to eliminate iterative elements, and the generated vulnerability signature will be particular to the attack instance and may have thousands of conditions if the attack instance is crafted with thousands of draw records. On the other hand, Tupni can identify not only fields but also record sequences and important constraints. Therefore, it can potentially be used to generate high-quality vulnerability signatures.

6. RELATED WORK

Network protocols and file formats have been subject to painstaking manual reverse engineering [18, 36]. Recently, there has been rising interest in automatic protocol reverse engineering.

The Protocol Informatics Project [35] and Discoverer [12] analyzed network traces to infer protocol formats. Fisher *et al.* [17] have presented a system that automatically infers the structure of an *ad hoc* data source and that produces a format specification in the PADS data description language. Specifically, the system infers several structural relationships based on the histogram of the frequencies of all tokens in the input. These systems are useful when a parsing program is unavailable. However, leveraging a parsing program can significantly improve the quality of the reverse engineered format.

Polyglot [6] uses dynamic data flow analysis to reverse engineer network message formats. With the rich information revealed by how a program consumes an input, Polyglot can identify basic fields and certain dependencies (e.g., length fields). Compared to Tupni, the main limitation of Polyglot is that it infers a message format as a “flat” sequence of fields, and cannot infer record sequences that are common in network protocol and file formats. Furthermore, Polyglot does not generalize a network message format over multiple message samples.

AutoFormat [26] reverse engineers network message formats by using context-aware monitored execution. To identify fields, it

groups contiguous inputs accessed consecutively in a function as a field. This heuristic works well for ASCII strings but is likely to fail for complex data structures. For example, different binary fields will be clustered into a single field if an application makes two subsequent accesses to adjacent input fields in a function. To identify record sequences, it identifies input positions that have been read by the same functions and declares each of them to be the start of a record. The input data between two subsequent record start markers is considered to belong to the first record. This heuristic may produce arbitrary false positives when confronted with complex input formats. For example, if an application accesses the first and the last byte of the input in the same functions, AutoFormat will conclude that the entire input is a record sequence. It also does not recognize nested record sequences such as those in WMF, JPG and PNG. AutoFormat does not attempt to identify record types or aggregate the analysis over multiple input samples.

In [42], Wondracek *et al.* propose a system for automatic network protocol analysis. When identifying basic fields, the system breaks an input chunk that cannot be attributed to any semantics (e.g., delimiters, lengths) into individual bytes and thus may fail to identify binary fields with more than one byte. For record sequences, the system can only identify those in which all records are structurally identical. Therefore, it cannot identify complex record sequences such as those in HTTP, WMF, JPG, PNG and TIF. It also does not identify record types. Unlike Polyglot and AutoFormat, the system in [42] can generalize a network message format over multiple messages. However, its sequence alignment algorithm may introduce more inaccuracies than Tupni because it does not have type information for individual byte fields and assumes any byte field can be matched with any other byte field. In contrast, Tupni infers field types based on the set of instructions that operate on each field and aligns fields based on their types.

Compared to previous efforts [6,26,42] that reverse engineer network message formats by observing how a program processes network messages, Tupni makes significant advances in the following ways: (1) It can identify *arbitrary* record sequences, including nested ones, by analyzing loops in a program; (2) It can infer field or record types by comparing the set of instructions that process each field or record; (3) It can derive constraints of various, not pre-defined dependencies across fields or messages by tracking symbolic predicates from dynamic data flow analysis; (4) It can improve its format specifications by aggregating its analysis over multiple inputs.

In parallel to our work, Lin and Zhang [27] developed two dynamic analysis techniques to derive syntactic structures of inputs with top-down or bottom-up grammars. The dynamic analysis on inputs with top-down grammars makes the assumption that pro-

gram control dependence follows the parsing structure. However, real-world programs may not follow this assumption (e.g., a program may scan an input multiple times). Unlike Tupni, this system does not attempt to aggregate the analysis over multiple inputs or infer input constraints.

Besides using dynamic analysis, Lim *et al.* [25] have proposed *FFE/X86*, a binary static analysis tool that infers file formats based on the program's output routines. The system can generate grammars with alternation and repetition, but requires manual annotations to the executable.

Previous work has also studied reverse engineering for specific applications such as replay. RolePlayer [13] and ScriptGen [23, 24] leverage byte-wise sequence alignment techniques to achieve application-level replay. Replayer [29] uses binary analysis to replay an application dialog. Kannan *et al.* semi-automatically discover session structures [22]. Christodorescu *et al.* perform binary static analysis to infer all possible strings at a given program point [8].

Dynamic data flow analysis has been used in many security applications, including exploit detection, patch generation, worm containment, and fuzz testing [10, 11, 14, 15, 19, 30, 31].

7. LIMITATIONS AND FUTURE WORK

In this section, we discuss the limitations of Tupni and describe future research directions for solving these limitations.

A fundamental limitation faced by Tupni is that the coverage of its reverse engineered format is dependent on the diversity of the input samples. We can mitigate this limitation from two perspectives. First, ShieldGen [14] and SAGE [19] have shown that one can construct new inputs to probe a program, which leads to better code coverage. Second, it is also possible to combine dynamic analysis with static analysis. However, it remains a research problem to integrate them effectively for reverse engineering a complete input format.

Another fundamental limitation in Tupni is that the constraints identified by Tupni may be specific to the input samples, which is a general limitation of dynamic analysis. This limitation can be mitigated by techniques presented in ShieldGen [14] and Bouncer [9].

Currently, Tupni does not work well for certain classes of input formats. The first class are formats whose field boundaries do not coincide with byte boundaries. This includes many encrypted and compressed file formats. The reason is that the Tupni prototype tracks taint at the byte level. We believe that this problem could be resolved by a more sophisticated implementation.

Another class of inputs that causes problems for Tupni are formats that cannot be described as regular expressions. For example, attempts to generate a C language specification by having Tupni analyze execution traces of a compiler would most likely produce poor results.

In addition, the quality of Tupni's output depends on the quality of the parsing application. Applications that neglect to compute checksums, ignore fields or deviate in other ways from the published specification reveal no information about those aspects of the format specification. This was the main cause of errors in our experiments. Any system that attempts to use a parsing application to obtain a format specification faces this problem. A possible way to mitigate this problem is to run multiple parsing applications on the input samples.

8. CONCLUSIONS

We have presented Tupni, a tool to reverse engineer input formats. Based on the observation that an application usually pro-

cesses iterative data records in a loop, Tupni can automatically identify (nested) record sequences, and record types in input formats. Tupni also can find different types of constraints on the values of fields. Furthermore, Tupni can generalize input formats over multiple inputs. We have demonstrated that Tupni can effectively reverse engineer 10 common, real-world file and network message formats. Given the rich set of information available in the input formats reverse engineered by Tupni, it can potentially be used for security applications such as zero-day vulnerability signature generation.

9. ACKNOWLEDGMENTS

We would like to thank Xuxian Jiang for his valuable comments on an early draft of this paper. We are grateful to Andrew Edwards, Stuart de Jong and Jinlin Yang for helping us understand iDNA. We also want to thank the anonymous reviewers for their constructive comments.

10. REFERENCES

- [1] International Telecommunication Union Recommendation T.81: JPEG Specification. <http://www.w3.org/Graphics/JPEG/itu-t81.pdf>.
- [2] TIFF Revision 6.0. <http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf>.
- [3] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.
- [4] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinic, D. Mihocka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the Second International Conference on Virtual Execution Environments*, Ottawa, Canada, June 2006.
- [5] N. Borisov, D. J. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo. A Generic Application-Level Protocol Analyzer and its Language. In *Proceedings of the 14th Annual Network & Distributed System Security Symposium (NDSS)*, March 2007.
- [6] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, Alexandria, VA, October 2007.
- [7] B. Chandra and M. Halldórsson. Greedy local improvement and weighted setpacking. *Journal of Algorithms*, 39(2):223–240, 2001.
- [8] M. Christodorescu, N. Kidd, and W.-H. Goh. String analysis for x86 binaries. *6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*, 31(1):88–95, 2006.
- [9] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing Software by Blocking Bad Input. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 117–130, 2007.
- [10] M. Costa, J. Crowcroft, M. Castro, and A. Rowstron. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP 2005)*, 2005.
- [11] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-37)*, Portland, OR, December 2004.

- [12] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic Protocol Reverse Engineering from Network Traces. In *Proceedings of the 16th USENIX Security Symposium*, Boston, MA, August 2007.
- [13] W. Cui, V. Paxson, N. C. Weaver, and R. H. Katz. Protocol-Independent Adaptive Replay of Application Dialog. In *Proceedings of the 13th Symposium on Network and Distributed System Security (NDSS 2006)*, February 2006.
- [14] W. Cui, H. J. Wang, M. Peinado, and M. Locasto. ShieldGen: Automated Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, May 2007.
- [15] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. 2007.
- [16] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. <http://www.w3.org/Protocols/rfc2616/rfc2616.html>.
- [17] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: Fully automatic tool generation from ad hoc data. In *Proceedings of the 35th Annual Symposium on Principles of Programming Languages*, January 2008.
- [18] Gaim Instant Messaging Client. <http://gaim.sourceforge.net>.
- [19] P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, February 2008.
- [20] T. O. Group. Dce 1.1: Remote procedure call. <http://www.opengroup.org/onlinepubs/9629399/toc.htm>.
- [21] E. Hazan, S. Safra, and O. Schwartz. On the complexity of approximating k -set packing. *Computational Complexity*, 15(1):20–39, 2006.
- [22] J. Kannan, J. Jung, V. Paxson, and C. E. Koksals. Semi-Automated Discovery of Application Session Structure. In *Proceedings of the 2006 Internet Measurement Conference (IMC)*, Rio de Janeiro, Brazil, October 2006.
- [23] C. Leita, M. Dacier, and F. Massicotte. Automatic Handling of Protocol Dependencies and Reaction to 0-Day Attacks with ScriptGen based Honeypots. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection*, Hamburg, Germany, September 2006.
- [24] C. Leita, K. Mermoud, and M. Dacier. ScriptGen: An Automated Script Generation Tool for Honeyd. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC 2005)*, December 2005.
- [25] J. Lim, T. Reps, and B. Liblit. Extracting file formats from executables. In S. E. Sim and M. Di Penta, editors, *Proceedings of the 13th Working Conference on Reverse Engineering*, Benevento, Italy, Oct. 23–27 2006.
- [26] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In *Proceedings of NDSS'2008*, 2008.
- [27] Z. Lin and X. Zhang. Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, November 2008.
- [28] P. Mockapetris. RFC 1035: Domain Names - Implementation and Specification. <http://www.faqs.org/rfcs/rfc1035.html>.
- [29] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic Protocol Replay by Binary Analysis. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, pages 311–321, 2006.
- [30] J. Newsome, D. Brumley, and D. Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2006.
- [31] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Symposium on Network and Distributed System Security (NDSS)*, February 2005.
- [32] R. Pang, V. Paxson, R. Somer, and L. Peterson. binpac: A yacc for Writing Application Protocol Parsers. In *Proceedings of the 2006 Internet Measurement Conference*, October 2006.
- [33] V. Paxson. Bro: A system for detecting network intruders in real time. *Computer Networks*, 31(23):2435–2463, 1999.
- [34] J. Postel and J. Reynolds. Rfc 959: File transfer protocol (ftp). <http://www.faqs.org/rfcs/rfc959.html>.
- [35] The Protocol Informatics Project. <http://www.baselineresearch.net/PI/>.
- [36] How Samba Was Written. http://samba.org/ftp/tridge/misc/french_cafe.txt.
- [37] K. Sollins. Rfc 1350: The tftp protocol (revision 2). <http://www.faqs.org/rfcs/rfc1350.html>.
- [38] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, October 2004.
- [39] T. Swan. *Inside Windows File Formats*. Sams Publishing, 1993.
- [40] T. Boutell, et. al. RFC 2083: PNG (Portable Network Graphics) Specification. <http://www.faqs.org/rfcs/rfc2083.html>.
- [41] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the ACM SIGCOMM*, August 2004.
- [42] G. Wondracek, C. Kruegel, E. Kirda, and P. Milani. Automatic Network Protocol Analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, February 2008.