P2C: Understanding Output Data Files via On-the-Fly Transformation from Producer to Consumer Executions

Yonghwi Kwon*, Fei Peng*, Dohyeong Kim*, Kyungtae Kim*, Xiangyu Zhang*, Dongyan Xu*,
Vinod Yegneswaran†, and John Qian‡
*Department of Computer Science, Purdue University
Email: {kwon58, pengf, kim1051, kim1798, xyzhang, dxu}@cs.purdue.edu
†SRI International
Email: vinod@csl.sri.com
‡Cisco Systems
Email: johnq@cisco.com

***Abstract*—In cyber attack analysis, it is often highly desirable to understand the meaning of an unknown file or network message in the *absence* of their consumer (i.e. the program that parses and understands the file/message). For example, a malware may stealthily collect information from a victim machine, store them as a file and later send it to a remote server. P2C is a novel technique that can parse and understand unknown files and network messages. Given a file/message that was generated in the past without the presence of any monitoring techniques, and a set of potential producers of the file/message, P2C systematically explores the execution paths in the producers without requiring any inputs. In the mean time, it tries to transform a producer execution to a consumer execution that closely resembles the ideal consumer execution that can parse the given unknown file/message. In particular, when a write operation is encountered in the original execution, P2C performs the opposite read operation on the unknown file/message and patches the original execution with the loaded value. In order to handle correlations between data fields in the file/message, P2C follows a trial-and-error approach to look for the correct transformation until the file/message can be parsed and the meaning of their fields can be disclosed. Our experiments on a set of real world applications demonstrate P2C is highly effective.**

I. Introduction

Understanding data files and network messages with unknown types or unknown structures is a prominent security challenge. Consider the following scenario: a user downloads and installs an Internet freeware that provides some desirable functionalities. After using the freeware for a period of time, she finds that some mysterious binary files are generated on her hard disk. While these may be normal data files used by the freeware, they could also contain private data such as personal profile, contact list, and even key-strokes that the freeware has stealthily collected from the victim machine. The consumer of these data files is often not present on the victim machine. In other words, there is not a program on the victim machine that can parse and understand these data files. Consider another scenario in which network monitoring is deployed on an enterprise network. Assume some machines are infected by an Advanced Persistent Threat (APT) malware, which stays inactive until the targeted attack date is getting close. That is the time the malware starts to communicate with the remote Command and Control (C&C) server to coordinate the attack. The network monitoring system picks up some packets to a rarely visited URL. It is highly desirable to understand the meaning of the packets. However, the consumer of the packets is on the remote site. The aforementioned two scenarios represent a prominent challenge: *Can we understand an existing file/message (generated in the past) with unknown format and unknown meanings in the absence of the consumer of the file/message*. This paper aims to address the problem.

There are a large pool of existing works on protocol reverse engineering [5], [12], [21], [6], [16], [4], [20] and input file format reverse engineering [12], [13]. These techniques work by monitoring the execution of consumer, that is, how the network messages/files are parsed and processed. For example, if consecutive bytes in an input file are accessed by consecutive instances of the same instruction, these consecutive bytes often constitute a buffer field. Most of these techniques focus on disclosing the syntactic structure of input files/messages. There are type reverse engineering techniques that can recover variable types, semantics and data structure definitions from stripped binaries [4], [14], [10], [19]. These techniques track data flow between variables and API functions with known interfaces. For example, if a variable has its value passed to a parameter in `strlen()`, this variable must be a string pointer. They can be combined with the aforementioned protocol/input-file-format reverse engineering techniques to derive both the syntax and semantics of input files/messages. However, the biggest limitation for most of these techniques is that they require the presence of the consumer. Some may not require the consumer [4], [20], but they require the user to provide inputs to drive producer execution (in order to apply dynamic analysis) and monitor execution that produced files/messages with the exact same format as the unknown file or message. In our scenario, the file/message was generated in the past without any monitoring. The user knows a set of potential producers but she does not know which one generated the file/message or which function of the producer generated it. She may not even know how to run the producer (e.g. a botnet malware without the presence of the C&C server may not run properly).

NDSS '15, 8-11 February 2015, San Diego, CA, USA
Copyright 2015 Internet Society, ISBN 1-891562-38-X
http://dx.doi.org/10.14722/ndss.2015.23318

Lim et al. proposed a technique that performs static analysis on producer binaries to derive the syntactic structure of an output file [11]. It generates a regular expression from program structure, which can be used to parse output files. For example, a file write inside a loop leads to a kleene closure in the resulting regular expression. However, the technique over-approximates output format due to the conservativeness of the underlying static analysis. It cannot denote constraints between fields such as a preceding field describing the length of the following field. It may generate regular expression such as "`(4 bytes)*(4 bytes)*`" to denote two consecutive buffer fields, not being able to express the constraints of/between the two buffers. Note that this regular expression can parse any output files. But the resulting parse tree unfortunately does not reflect the real structure. Furthermore, the technique cannot associate meanings to data fields.

In [8], Driscoll et al. have a nice observation that producer and consumer are symmetric. Although the technique has a different goal, which is to statically verify the correctness of a producer by checking its conformance to the corresponding consumer or vice versa, the observation inspired our solution, called P2C. The basic idea of P2C is the following. Given a file[1] that we want to understand and a potential producer binary, we will leverage a recent binary forced-execution technique [17] to explore the paths of the binary, by concretely executing the producer along a large number of paths *without* requiring any inputs. When a file open-for-write operation is encountered, the operation may be the place where the subject file was created, P2C starts to monitor the producer execution and transform it to the corresponding consumer execution. In particular, when a file write is encountered, it performs the opposite read on the subject file. The values that are read are patched to the execution such that the following operations can be performed properly, in a way closely resembling the ideal consumer execution. For example, by transforming a *buffer size write* to the corresponding *buffer size read* from the subject file and setting the size variable accordingly in the execution, when the following *buffer write* is transformed to the corresponding *buffer read*, the proper number of bytes are read. Once the file can be correctly parsed, data flow tracking based reverse engineering technique can be used to associate semantics to the fields in memory.

Our contributions are summarized in the following.

- We propose the novel idea of on-the-fly transformation from a producer execution to a consumer execution to understand unknown files/messages that were generated in the past, without the presence of the real consumers. In fact, we find that in some cases, the transformed consumers are even better than the real consumers as the real ones sometimes skip data fields. The technique does not require the user to know the precise producer or how to run a potential producer binary.
- We identify that the prominent technical challenge lies in handling correlations between fields. In some cases, such fields correlations are not sufficiently exposed from the producer logic. As a result, preceding symmetric reads in a transformed execution may not provide enough guidance for the following correlated symmetric reads. We have formally described the nature of the problem and proposed an iterative search based algorithm to address the problem.
- We have evaluated P2C on a variety of real world binary programs. Our results show that P2C is able to correctly parse their output files or messages and disclose the correct meanings for most fields.

II. Demonstrative Example

In this section, we use an example to illustrate P2C. The code snippet of the example is presented in Fig. 1. The program allows users to create accounts and save account information to a file. Given a data file that was generated previously and the program itself, our goal is to understand the file, including identifying individual data fields and their types and meanings. Assume the program only serves as the producer of data files and it does not have the consumer (parsing) logic. As such, existing input format reverse engineering techniques [5], [12], [13] that rely on observing how a file is parsed cannot be applied.

As shown in Fig. 1, the program has two related data structures. Struct `header` represents the header information of the output file, which contains the type of the accounts (e.g., type 0, 1, and 2, denoting different permissions), the number of accounts, the checksum of the file, and the name of the group of accounts in this file (e.g., "`basic`" and "`silver_member`"). Struct `account_entry` represents the information for each account, including the id, the creation date, and the balance. Method `create_account()` is used to create accounts. It first allocates space for a new account and saves it to the `accounts[]` array (line 22). It also initializes the creation date related fields by calling system API functions (lines 23, 24 and 26). It then sends an email using the newly created id to notify the account creation (line 28). Method `output()` emits account information to a file. It first emits a magic number (line 45). It then sets the header fields (lines 46-47). It computes the size of the header (line 48), which is dependent on the length of the `name` field in the `header` struct. It emits the header size (line 49), followed by the header (line 50). After that, it uses a loop (lines 51-52) to emit the individual accounts.

Assume in a previous native execution of the program, an output file was emitted as Fig 2 (c). Now we want to understand the structure of the file and the semantics of the individual fields of the file.

We first need an effective execution path exploration tool to identify all the output file creation operations (e.g. `fopen()` for writes). We decide to leverage a binary forced-execution platform we recently developed, called X-Force [17], to meet this requirement.

X-Force executes an x86 binary program without requiring any inputs. It supplies random inputs when needed. It has an exception recovery scheme that suppresses runtime exceptions. For example, it allocates memory on-demand when an invalid pointer is de-referenced. At runtime, it also systematically

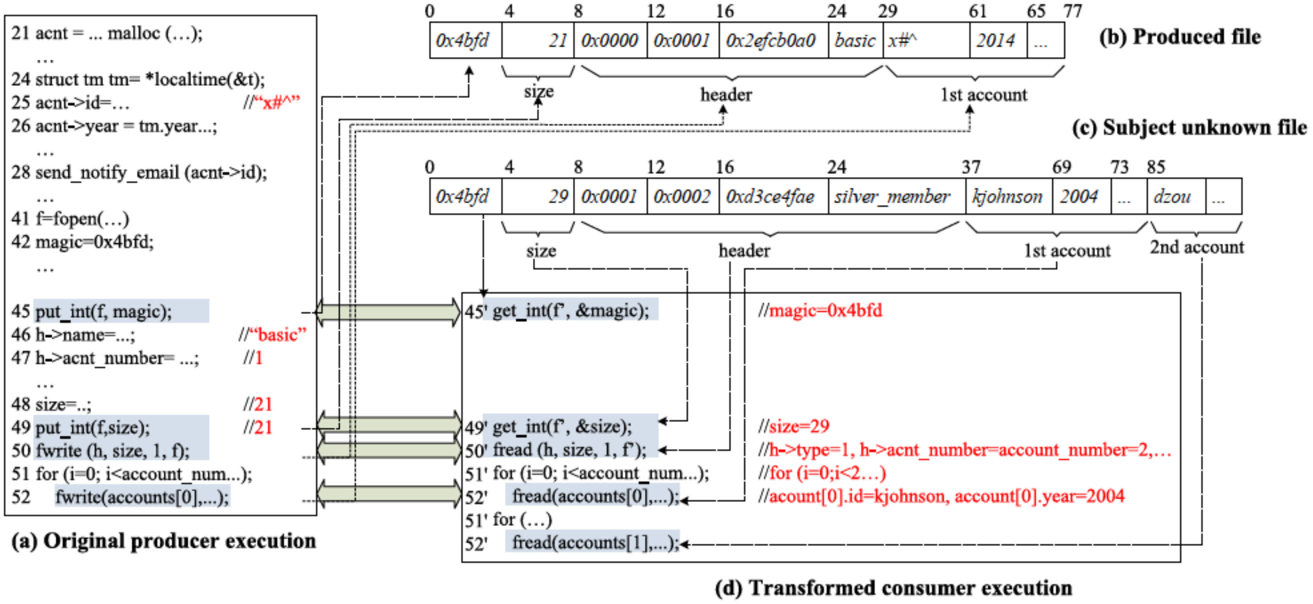
[1] While P2C works on both files and network messages, we will focus our discussion on files. Handling messages is similar.

```
1   typedef struct header {
2     int type;
3     int acnt_num;
4     long checksum;
5     char name[128];
6   } header;
7
8   typedef struct account_entry {
9     char id [32];
10    int year;
11    int month;
12    int day;
13    int balance;
14  } account_entry;
```

```
20 void create_account (…) {
21     account_entry * acnt = ... malloc (…);
22     acounts[account_num++]=acnt;
23     time_t t= time (NULL);
24     struct tm tm= *localtime(&t);
25     acnt->id=…
26     acnt->year = tm.year...;
27     …
28     send_notify_email (acnt->id);
29     …
30 }
```

```
40  void  output(accunt_entry** accounts) {
41     FILE * f = fopen(…);
42     int magic=0x4bfd;
43     header * h = (header*) malloc(sizeof(header));
45     put_int(f, magic);
46     h->name=strcpy(…);
47     h->acnt_number= account_num;
48     size=sizeof(header)-128+strlen(h->name);
49     put_int(f, size);
50     fwrite (h, size, 1, f);
51     for (i=0; i<account_num; i++);
52         fwrite(accounts[i], sizeof(account_entry),1,f);
53  }
```

Fig. 1. The code snippet of the demonstrative example.


```
21 acnt = ... malloc (…);
   …
24 struct tm tm= *localtime(&t);
25 acnt->id=…                //"x#^"
26 acnt->year = tm.year...;
   …
28 send_notify_email (acnt->id);
   …
41 f=fopen(…)
42 magic=0x4bfd;
   …

45 put_int(f, magic);
46 h->name=...;              //"basic"
47 h->acnt_number= ...;      //1
   …
48 size=..;                  //21
49 put_int(f,size);          //21
50 fwrite (h, size, 1, f);
51 for (i=0; i<account_num...);
52    fwrite(accounts[0],...);
```

(a) Original producer execution

```
45' get_int(f', &magic);             //magic=0x4bfd



49' get_int(f', &size);              //size=29
50' fread (h, size, 1, f');          //h->type=1, h->acnt_number=account_number=2,…
51' for (i=0; i<account_num...);     //for (i=0;i<2…)
52'    fread(accounts[0],...);       //acount[0].id=kjohnson, account[0].year=2004
51' for (…)
52'    fread(accounts[1],...);
```

(d) Transformed consumer execution

Fig. 2. Understanding an output file generated in the past by transforming the current (producer) execution. The file operations are highlighted in the traces. The relevant states are shown on the right of each trace. The dashed arrows associate a file operation with the corresponding field in the file. The field offsets are presented on the top.

force-sets the branch outcomes of a bounded number of predicates (out of the hundred of thousands predicate executions in a single run) to explore a large number of execution paths. It was used to reverse engineer variable types and data structure definitions in stripped binaries by observing data flow between variables and library/system calls, whose parameter types and semantics are known before-hand. For example, if a variable is copied from the parameter of `strcpy`, X-Force can infer that its type is `char*`. □

In our usage scenario, when X-Force identifies an output file creation operation, the current execution and the operation are passed to P2C to start the transformation to parse the given unknown file. A failure is reported if all output file creation operations have been tried. Then a different producer candidate will be tested.

Fig. 2 (a) shows a trace of the program in Fig. 1 when it executes in X-Force. In the trace, the output file creation is encountered (line 41). In this execution, an account is created and inserted to the `accounts[]` array (lines 21-28). Since inputs are randomly generated (in X-Force), the `id` field in the account has a random value "`x#^`" (line 25). The `name` field in header `h` has the value "`basic`" (line 46). As a result, the header size is 21 (line 48). Finally, the execution produces a file as shown in Fig. 2 (b).

In order to transform the producer execution to parse the intended file (Fig. 2 (c)), when a file output operation is encountered, P2C *performs the opposite input operation on the subject unknown file*, which is also called the *symmetric read*. Particularly, the first file output operation in the original execution is to emit the magic number (line 45). Instead of performing the write operation `put_int()`, P2C performs `get_int()` on the subject file (line 45' in the transformed trace in Fig. 2 (d)). Having observed that the value to be emitted (in the original run) and the value read (in the transformed run) match, P2C proceeds to the next file operation. Similarly, the second write at line 49 is transformed to a read at line 49', which replaces the value of `size` to 29 from the subject file. Consequently, the symmetric read at line 50' reads 29 bytes instead of 21 in the original execution, which correctly loads the header in the subject unknown file. Similarly, the read at 50' replaces the fields in the header structure with the values loaded from the subject file. Now the number of accounts changes from 1 to 2. As a result, the transformed execution loads two account entries. Observe at the end, the transformed execution properly parses the entire subject file.

Next, P2C leverages the type reverse-engineering capability in X-Force to associate types and semantic tags to the fields from the subject file. For example, in the original execution, the `year` field of `accounts[0]` emitted at line 52 has data flow from the library call `localtime()` at line 24. This allows us to type the `year` field of `accounts[0]` in the original execution, which in turn types the `year` fields of `accounts[0]` and `accounts[1]` at line 52' through the symmetry between the original execution and the transformed execution. Similarly, the data flow between lines 25 and 28 about the `id` field indicates this field contains an email id. The data flow between 25 and 52 further allows P2C to determine that the `id` fields of `accounts[0-1]` loaded at 52' (in the transformed execution) are also email ids.

Observe that through such a transformation procedure, P2C is able to identify and type the fields from the file in Fig. 2 (c), which is not the file generated by the original execution (i.e. Fig. 2 (b)).

We want to point out that there are three observations critical to the effectiveness of P2C.

- The producer and consumer logics of the same kind of data files have symmetric structures. The write operations that generate a data file have the corresponding read operations in the consumer. *The essence of P2C is to leverage the control flow in the producer to generate the sequence of write operations such that the corresponding reads can be performed to parse the subject file.* We will discuss how P2C can handle rare cases in which producers and consumers are asymmetric in Section V.
- A data file is often self-contained, meaning that it has enough information to allow itself to be parsed. For example, the number of data entries must be present in the same file as an entry. Intuitively, one can imagine that if such a property did not hold, the corresponding consumer would have difficulty parsing the file independently. Later in Section V, we will discuss some exceptions, such as encrypted files.
- There are often dependences between fields, such as that between the header size field and the header field in Fig. 2 (b) and (c). We observe that the spacial order of these fields in the file often follows the dependence order. For example, the header size field precedes the header field while the parsing of the header field relies on the size field. In other words, the values loaded earlier often provide sufficient direction for later file operations. The property allows P2C to gradually transform the producer to the corresponding consumer. In our example, loading the `size` field at line 49' provides sufficient direction to load the `header` field at line 50', which in turn guides the loading of the two account entries.

III. Technical Challenges

We used an example in the previous section to demonstrate the basic idea of P2C. In the real world, producers are often much more complex and difficult to transform. According to our experience, these challenges root at the correlations between fields in the file. In particular, field correlations may not be sufficiently exposed in the producer logic such that the symmetric read of a preceding field may not provide sufficient guidance to the later read(s) of the dependent field(s) (in the transformed execution). Before we elaborate on this issue, we would like to first define the different kinds of field correlations. Based on their characteristics, we introduce the following two kinds of correlations: *data correlation* and *control correlation*.

Definition 1. *Two fields in a file are data-related if they are computed from some common data source in the producer execution that generated the file.*

Since the two fields share some common data source, which may be a variable or a data structure, any perturbation on the value of the source may cause both fields to change. Fig. 3 (e) shows such an example, in which both fields `v1` and `v2` have data flow from `data`.

Definition 2. *A field f_1 is control-related to another field f_2 if the value of f_2 directly decides the proper parsing of f_1.*

For example in Fig. 3 (a) and (b), the `buf` field is control-related to the preceding `len` field. Fig. 3 (c) and (d) show another kind of control correlation. In particular, field `id/l_id` is control-related to the preceding `type` field, which determines if an integer or a long value should be loaded from the subject file. Note that according to the definition, the correlation has to be direct. If f_3 is data-related to f_2 and f_2 directly decides the proper parsing of f_1, f_1 is not control-related to f_3.

In P2C, detecting and handling control correlations is much more important than data correlations because the former determines if P2C can properly parse the file fields. Note that once these fields are parsed into memory of the execution, data flow tracking based type reverse engineering can be leveraged to understand the meaning of the fields. In contrast, data correlations do not affect the parsing of fields. For example, in Fig. 3 (e), despite the data correlation, the two fields can be properly loaded when the symmetric reads corresponding to lines 42 and 46 are performed. Hence, our discussion will focus on control correlation in the rest of the paper.

We observe that some control correlations are easier to handle compared to others. In Fig. 3 (a), in the transformed execution, the symmetric read corresponding to line 6 would properly set the value of `len`, which allows the following symmetric read of `buf` to proceed properly. In contrast in Fig. 3 (b), although the symmetric read at line 15 sets the value of `len`, the following read of `buf` is incorrect because `strlen(buf)`'s value is still a random value depending on the content of `buf` in the original execution. For example, assume `buf`="`##`" in the original execution and the value in the subject unknown file is "`Yes`". Despite that the symmetric read of `len` (at line 15) gets the correct value of 3, the following read of `buf` will unfortunately still use the old incorrect length of 2 (computed from `strlen(buf)`). Observe that in both cases, the buffer field is control-related to the length field in the file.

Similarly, in Fig. 3 (c), the symmetric read at line 22 will drive the execution to take the proper branch at line 23 such

```
1  f= fopen (…);
2
3  len = strlen(buf);
4  …
5
6  put_int(len);
7  fwrite(buf, len,1,f);
```

(a)

```
10 f= fopen (…);
11
12 len = strlen(buf);
13 …
14
15 put_int(len);
16 fwrite(buf, strlen(buf),1,f);
```

(b)

```
20 f= fopen (…);
21 type=...;
22 put_int(type);
23 if (type==1)
24    put_int (id);
25 else
26    put_long (l_id);
```

(c)

```
30 f= fopen (…);
31 type=foo(header);
32 put_int(type);
33 if (foo(header)==1)
34    put_int (id);
35 else
36    put_long (l_id);
37 fwrite(header, 32, 1, f);
```

(d)

```
40 f= fopen (…);
41 v1=foo(data);
42 put_int(v1);
43
44 …
45 v2=gee(data);
46 put_int (v2)
```

(e)

Fig. 3. Examples for different kinds of field correlations; (a) and (c) are exposed control correlation; (b) and (d) are unexposed control correlation; (e) is unexposed data correlation.

that the correct following symmetric read will be performed. In contrast, in Fig. 3 (d), reading the correct value of `type` from the file does not help choosing the correct branch.

The root cause lies in whether control correlations between file fields manifest themselves through program dependences. We hence further divide field correlations to the following two classes.

Definition 3. *A field control-correlation, f_1 correlated to f_2, is* exposed *if there is a program dependence path from f_2 to f_1 in the producer execution that generated the file.*

If there is not such a dependence path, we say the field correlation is *unexposed*. Observe that in Fig. 3 (a), the field control correlation is exposed by the program dependence edge from 3 to 7 (by variable `len`). In contrast in (b), there is not a program dependence path from 12 to 16. Note that although at line 12, `len` is computed from `buf`, the data flow does not expose the field correlation in the opposite direction, which is the `buf` field control-related to the `len` field. As a result, loading the correct value of `len` at 15 does not help the following read of `buf` at 16 as the `len` value cannot propagate to and affect line 16 along program dependences.

In (c), the program dependence path 21 → 23 → 24/26 exposes that the `id` field is control-related to the `type` field. In contrast, the correlation in (d) is not exposed.

A key challenge to P2C is hence to handle unexposed field control correlations (e.g., Fig. 3 (b) and (d)).

IV. Design

$$
\begin{array}{lll}
\textit{Program} & P ::= & s \\
\textit{Stmt} & s ::= & s_1; s_2 \mid \texttt{skip} \mid r :=^{\ell} e \mid r :=^{\ell} \texttt{R}(r_a) \mid \\
 & & \texttt{W}^{\ell}(r_a, r_v) \mid \texttt{goto}^{\ell}(\ell_1) \mid \texttt{if}\ (r^{\ell})\ \texttt{then}\ \texttt{goto}(\ell_1) \mid \\
 & & r_f := \texttt{fopen4write}^{\ell}(r_n) \mid \texttt{fwrite}^{\ell}(r_{buf}, r_l, r_f) \mid \\
 & & \texttt{fclose}^{\ell}(r_f) \mid \texttt{call}^{\ell}(\ell_1) \mid \texttt{ret}^{\ell}(c) \\
\textit{Operator} & op ::= & + \mid - \mid * \mid / \mid ... \\
\textit{Expr} & e ::= & r \mid c \mid a \mid r_1\ \texttt{op}\ r_2 \\
\textit{Register} & r ::= & \{r_1, r_2, r_3, ...\} \\
\textit{Const} & c ::= & \{true, false, 0, 1, 2, ...\} \\
\textit{Addr} & a ::= & \{0, 1, 2, ...\} \\
\textit{Label} & \ell ::= & \{\ell_1, \ell_2, \ell_3, ...\}
\end{array}
$$

Fig. 4. Language

In this section, we discuss the design of P2C, focusing on handling unexposed control correlations between fields.

Language. P2C works on x86 binary executables, without requiring any symbolic information. However, if we were to discuss the design at the x86 level, the complexity of the instruction set would render the discussion and the example presentation very verbose. Hence, we introduce a simplified low level language, in which we only model enough to present our technique. The language is presented in Fig. 4[2].

Memory reads and writes are denoted by $\texttt{R}(r_a)$ and $\texttt{W}(r_a, r_v)$, respectively, with r_a holding the address and r_v the value. As a low level language, conditional or loop statements are not modeled. Instead we define jumps using `goto` and guarded `goto`. We only model a small number of system calls that are file output related: `fopen4write()` takes r_n as a pointer to the buffer of a file name, creates the file for write, and returns the handler in r_f; `fwrite()` writes the content in the buffer pointed to by r_{buf} with length r_l to file r_f. In the rest of the paper, we will also use output operations `put_int()` and `put_long()` for brevity, although they are not part of the language. Note that translating them to `fwrite()` operations is straightforward. Other function invocations and returns are denoted by `call`(ℓ) and `ret`, respectively. The return value of a function call is stored in r_0. Each statement is annotated with a label, which can be intuitively considered as the program counter (PC) on x86.

Design Overview. Due to the presence of unexposed field control correlations, simply transforming a producer execution to the corresponding consumer execution may not work. Therefore, P2C is essentially an iterative procedure that searches for a valid transformation to parse the file. In particular, when it encounters file output operations in which the symmetric input operations cannot be determined with certainty (due to unexposed correlations), P2C would select one option to proceed. For example in Fig. 3 (b), at line 16, the length of file write, which is also the length of the symmetric read, does not come from a value loaded from the file, but rather a value computed from the current state of `buf` in the original execution. In other words, P2C does not know how many bytes it should read. In this case, P2C will pick a value, which may be the current length value `strlen(buf)` to proceed. To detect in-appropriate choices, after each symmetric file read operation (in the transformed run), P2C will re-execute from the file open operation (i.e. the starting point of P2C) to detect any inconsistency, which will cause P2C to roll back to pick another option to explore at one of the uncertain points, until either the file is properly parsed or all options have been explored. In this case, P2C will test the next producer candidate.

In Fig. 3 (b), assume the file to be parsed has a length value 3 followed by a string “`Yes`”. The length value 3 is properly read at line 15 through the symmetric read. Since the following operation at line 16 has a length value not from the file, P2C creates a backtrack point and then selects

[2] Since we leverage X-Force as our execution engine, the language here bears some resemblance to that defined in [17].

the current `strlen(buf)` to proceed. Assume the current `strlen(buf)`=2, the symmetric read only loads "`Ye`" to the buffer. Re-executing with the length value and the buffer value loaded from the file discloses an inconsistence. In particular, at line 12, the left-hand-side `len` should have value 3 (according to the file), but the right-hand-side `strlen()` operation on the buffer loaded from the file has a value 2. As such, P2C rolls back and selects a different length value at the previous backtrack point at line 16. This time, it increases the length value by one and reads 3 bytes. Re-execution does not detect any inconsistency. P2C admits the choice and proceeds.

$$\begin{array}{lll}
\mathit{Store}\ \sigma & ::= & (\mathit{Addr}\ \mid\ \mathit{Register}) \rightarrow \mathit{Const} \\
\mathit{TaintStore}\ \tau & ::= & (\mathit{Addr}\ \mid\ \mathit{Register}) \rightarrow \mathit{Taint} \\
\mathit{InstrCnt}\ \mathit{cnt} & ::= & \mathit{Label} \mapsto \mathit{Instance} \\
\mathit{OriginalDef}\ S & ::= & (\mathit{Addr}\ \mid\ \mathit{Register}) \rightarrow (\mathit{Label} \times \mathit{Instance}) \\
\mathit{Trace}\ T & ::= & \overline{E} \\
\mathit{TraceEntry}\ E & ::= & \langle \ell, i, c, t \rangle \\
\mathit{Instance}\ i & ::= & \mathbb{Z} \\
\mathit{Taint}\ t & ::= & \mathit{PATCH\_RELATED}\ \mid\ \mathit{INPUT\_RELATED}\ \mid \\
 & & \mathit{NONE} \\
\mathit{Patch}\ P & ::= & (\mathit{Label} \times \mathit{Instance}) \rightarrow \mathit{Constant}
\end{array}$$

Fig. 5. Definitions.

A. *Algorithm*

In this subsection, we will present the algorithm. Later subsections explain the different sub-components of the algorithm.

Definitions for Algorithms and Semantic Rules. To discuss the algorithm and its components, we introduce a few definitions in Fig. 5. Store σ is a mapping from a memory address or a register to a value, representing regular program state. Taint store τ associates each address/register with a taint that may be (1) *PATCH_RELATED*, denoting that the current value in the address/register is computed from a value brought in by a symmetric read (called a *patch*); (2) *INPUT_RELATED*, denoting the current value is computed from some input that is not a patch. Since the original execution is from X-Force, such input values are random; (3) *NONE*, denoting the other cases. Note that we need tainting to determine if an operation is uncertain such that a backtrack point needs to be created, and to detect inconsistency during re-execution. Instruction count *cnt* denotes how many instances a statement has been executed. This is to uniquely identify an execution instance of a statement. P2C also collects trace for inconsistency detection. A trace T consists of a sequence of trace entries E, which is a four-value tuple with the label (i.e. PC), the execution instance, the left hand side value of the statement, and the taint of the left hand side value.

The original definition map S is a mapping from an address/register to an instruction instance that denotes the original definition point of the value held in the address/register. This is to facilitate patch generation. Note that when a variable is emitted at an output point, the symmetric read operation will bring in the corresponding value in the file. But the value should be set at the original definition point of the variable instead of at the current file output point. For example, consider the following code sequence:

```
1 type=foo(data); 2 x=type; 3 y=x; 4 put_int(y)
```

[C1]

The symmetric read will load the value at line 4, but we should patch the value of `type` at line 1. The benefit is that all the related variables: `x`, `y`, and `type` are correctly patched when we re-execute the code from the patched line 1. The original definition map is to facilitate such patching. In practice, file output system calls often occur inside library functions and use internal buffers. Without identifying the original definitions that are usually in the user code, symmetric file reads only patch the values of internal buffers and cannot influence user code execution.

Patch P maps an instruction instance to a value, denoting the patch for that instance. □

Algorithm 1 Iterative Parsing Procedure

```
    Input: Store $\sigma_0$: the starting store.
           TaintStore $\tau_0$: the starting taint store.
           Label $\ell_0$: the starting PC.
           $F$: the subject file to parse.
    Output: a patch or NOT-PARSABLE
    Definition: OriginalDef $S$: the mapping from an address/register to its origin definition
           Patch $P$: the current patch.
 1: function PARSEFILE($\sigma_0$,$\tau_0$, $\ell_0$,$F$)
 2:   $\langle T,S,\tau,\sigma,\ell\rangle \leftarrow \langle nil,nil,\tau_0,\sigma_0,\ell_0\rangle$
 3:   $P \leftarrow nil$
 4:   while true do
 5:     if ISFILECLOSE($\ell$) then
 6:       if ISEOF($F$) then
 7:         return $P$
 8:       else
 9:         if there are unexplored options then
    /*Roll back and explore a new selection*/
10:           BACKTRACK();
11:         else
12:           return NOT-PARSABLE
13:         end if
14:       end if
15:     end if
16:     $\langle T', S', \tau', \sigma', \ell'\rangle \leftarrow$ execute from $\langle T, S, \tau, \sigma, \ell\rangle$ to the next file output operation with patch $P$
    /*Perform symmetric input operation and update patch*/
17:     UPDATEPATCH($S'$, $\tau'$, $\sigma'$, $\ell'$, $F$)
    /*Re-execute with the new patch and validate consistency*/
18:     $\langle T'', S, \tau, \sigma, \ell\rangle \leftarrow$ execute from $\langle nil, nil, \tau_0, \sigma_0, \ell_0\rangle$ to the same output operation with the updated patch $P$
19:     if $\neg$ CHECKCONSISTENCY($T'$, $T''$) then
20:       if there are unexplored options then
21:         BACKTRACK();
22:       else
23:         return NOT-PARSABLE
24:       end if
25:     end if
26:     $T \leftarrow T''$
27:   end while
28: end function

29: function UPDATEPATCH($S'$, $\tau'$, $\sigma'$, $\ell'$, $F$)
30:   Let the file operation at $\ell'$ be fwrite$^{\ell'}(r_{buf}, r_s, r_f)$
31:   if $\tau'[r_s] \equiv INPUT\_RELATED$ then
32:     $size \leftarrow$ select a value in the possible range of $r_s$
33:   else
34:     $size \leftarrow \sigma'[r_s]$
35:   end if
36:   $buf \leftarrow \sigma'[r_{buf}]$
    /*Perform the symmetric read, backtrack on exception*/
37:   fread($buf, size, 1, F$)
    /*Update the patch*/
38:   for $i$=0 to $size$ do
39:     $\langle \ell_p, i_p\rangle \leftarrow S'[buf + i]$
40:     $P[\langle \ell_p, i_p\rangle \mapsto buf[i]\ ]$
41:   end for
42: end function
```

The algorithm is presented in Algorithm 1. It is invoked when a file creation (for write) operation is encountered during path exploration in X-Force. It takes as input the store, the taint store, the label for the file creation operation, and the unknown subject file. It aims to transform the original execution between the file creation and the corresponding file close to parse the subject file. If it successfully parses the subject file, the algorithm will return the generated patch that decides where the values from the file are applied to various execution points. Otherwise, it indicates failure. The patch can be further leveraged to type the file fields.

The main procedure is defined in PARSEFILE(). The configuration of program execution in P2C is a tuple of five fields: the trace T, the original definition map S, the taint store τ, the store σ, and the label of the current statement ℓ. One can consider a program execution in P2C is a procedure of updating the configuration step by step. In later subsections, we will explain in the semantic rules how the different configuration fields are updated.

At line 2, the configuration is initialized. Note that the initial trace and the initial original definition map are set to *nil*. The WHILE loop in lines 4-27 represents the main iterative process. The algorithm progresses to a new file output operation in each iteration until the file close operation is encountered. In this procedure, uncertain selections may be backtracked. Inside the loop, the algorithm first determines if the file close operation is reached. If in the mean time, the subject file also reaches its end, the file is successfully parsed and it returns the current patch (lines 5-7). If there is still left-over in the subject file, the current parsing efforts are not successful, the algorithm will backtrack and try a different option in one of the earlier selection points, until all options are explored (lines 9-14).

If the file close operation has not been reached, the algorithm executes the program until the next file output operation is reached (line 16). It then invokes method UPDATEPATCH() to perform the symmetric file read and generate the corresponding patch (line 17). The patch is applied by re-executing from the initial file creation operation (line 18). The re-execution generates a new trace T'', which is compared with the previous trace T' (for the previous execution without applying the new patch) (line 19). If they are inconsistent, which must be caused by unexposed field control correlations, the algorithm backtracks (lines 19-25). The WHILE loop then starts another iteration to proceed to the next file output operation.

Function UPDATEPATCH() performs the symmetric file read and updates the patch with the newly loaded value(s). It first tests if the length value of the file write is tainted as $INPUT\_RELATED$ (line 31), meaning that it is not computed from a patched value, but rather from an input supplied by X-Force randomly. If so, it selects a value from the legal range of the length field, which is by default between 0 and the size of the entire file[3]. Note that the selection can be backtracked and neighboring values will be explored. If the length value is not tainted as $INPUT\_RELATED$, it is computed from either a patched value or a constant value. In this case, the algorithm uses the current length value (line 34). After the file read (line 37), the patch is updated by looking up the original definition for each byte in the buffer field and setting the left-hand-side values at those definitions to the ones loaded from the file. Note that if the symmetric read fails because the end of the subject file is reached, the algorithm also backtracks.

```
 1  //buf[]=fread(…);                    //put_int(len)
 2  r1= fopen4write (…);             7  r3=4;
                                     8  fwrite(r2, r3, r1);
    //len=strlen(buf)
 3  r0= buf;        //parameter         //fwrite(buf, strlen(buf),1,f);
 4  r0=call strlen;                  9  r0= buf;        //parameter
 5  r2= &len;                       10  r0=call strlen;
 6  W (r2, r0);                     11  r2= buf;
                                    12  fwrite (r2,r0,r1);
```

Fig. 6. The code snippet in our language corresponding to the example in Fig. 3 (b).

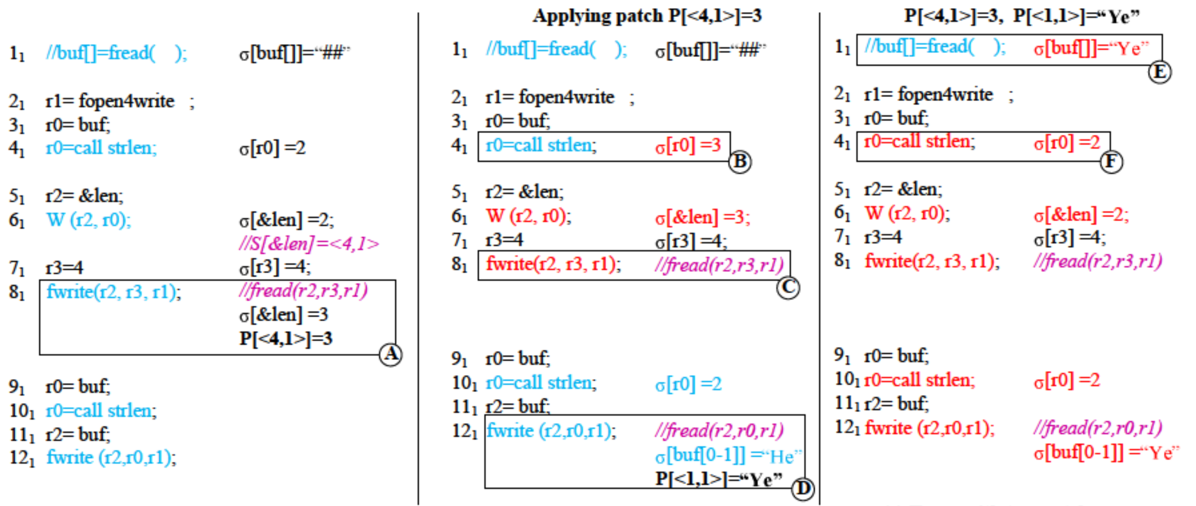
Example. We translate the example in Fig. 3 (b) to our language. The resulting code is shown in Fig. 6. Note that the parameter of the `strlen()` call is passed through register r_0 and the result of the call is also stored in r_0. Fig. 7 shows the execution of the algorithm on the example. It shows the traces of three executions, including the initial execution and two re-executions. For each trace, the control flow is shown on the left and the states are shown on the right.

Assume the subject file contains length value 3 followed by string "`Yes`". In the initial execution (Fig. 7 (a)), the buffer contains a random string "`##`" provided by X-Force. Observe that after the memory write to address $\&len$, the original definition of the address is updated to $\langle 4, 1\rangle$ (i.e. the first instance of line 4), which is the return site of `strlen()`. Next, we will explain the algorithm execution procedure by highlighting a few steps.

- At Ⓐ, a symmetric read with a deterministic length of 4 bytes loads the length value 3 from the file and the patch is to set the original definition of address $\&len$ to 3 (according to line 17 in the algorithm).
- The algorithm re-executes with the patch (according to line 18 in the algorithm), at Ⓑ, the patch is applied and the following instructions dependent on r_0 are tainted as $PATCH\_RELATED$.
- When the re-execution reaches Ⓒ, the algorithm does not detect any patch inconsistency by comparing trace (b) with trace (a)[4] (according to line 19 of the algorithm). As such, it tentatively admits the patch and proceeds to the next file operation at Ⓓ.
- At Ⓓ, the algorithm detects the length value is from a random value. It selects a value (i.e. value 2) to proceed and loads string "`Ye`" (according to lines 31-32 in the algorithm). Since `buf` was defined earlier than the starting point of the algorithm, we denote its definition point as a special value 1_1.
- The algorithm re-executes again with the new patch. It applies the patch to `buf` at Ⓔ.
- However, at Ⓕ, the patch applied on `buf` produces the length value 2, which is inconsistent with the other patch on `r0`. The algorithm backtracks, and resets the length value to 3 at Ⓓ. This time, the correct

[3] The first selection is the current length value.

[4] While we will explain consistency checking in later subsections, the intuition is that multiple patches cannot produce inconsistent values at an instruction.



```
1_1  //buf[]=fread(   );      σ[buf[]]="##"

2_1  r1= fopen4write   ;
3_1  r0= buf;
4_1  r0=call strlen;          σ[r0] =2

5_1  r2= &len;
6_1  W (r2, r0);              σ[&len] =2;
                              //S[&len]=<4,1>
7_1  r3=4                     σ[r3] =4;
8_1  fwrite(r2, r3, r1);      //fread(r2,r3,r1)
                              σ[&len] =3
                              P[<4,1>]=3

9_1  r0= buf;
10_1 r0=call strlen;
11_1 r2= buf;
12_1 fwrite (r2,r0,r1);
```

(a) Trace

Applying patch P[<4,1>]=3

```
1_1  //buf[]=fread(   );      σ[buf[]]="##"

2_1  r1= fopen4write   ;
3_1  r0= buf;
4_1  r0=call strlen;          σ[r0] =3

5_1  r2= &len;
6_1  W (r2, r0);              σ[&len] =3;
7_1  r3=4                     σ[r3] =4;
8_1  fwrite(r2, r3, r1);      //fread(r2,r3,r1)

9_1  r0= buf;
10_1 r0=call strlen;          σ[r0] =2
11_1 r2= buf;
12_1 fwrite (r2,r0,r1);       //fread(r2,r0,r1)
                              σ[buf[0-1]] ="He"
                              P[<1,1>]="Ye"
```

(b) Trace with one patch

P[<4,1>]=3, P[<1,1>]="Ye"

```
1_1  //buf[]=fread(   );      σ[buf[]]="Ye"

2_1  r1= fopen4write   ;
3_1  r0= buf;
4_1  r0=call strlen;          σ[r0] =2

5_1  r2= &len;
6_1  W (r2, r0);              σ[&len] =2;
7_1  r3=4                     σ[r3] =4;
8_1  fwrite(r2, r3, r1);      //fread(r2,r3,r1)

9_1  r0= buf;
10_1 r0=call strlen;          σ[r0] =2
11_1 r2= buf;
12_1 fwrite (r2,r0,r1);       //fread(r2,r0,r1)
                              σ[buf[0-1]] ="Ye"
```

(c) Trace with two patches

Fig. 7. The algorithm execution on the example in Fig. 6. The subject file to parse contains a length value 3 followed by a string "Yes". Label 6_1 denotes the first instance of line 6. The instructions and states in blue are tainted with *INPUT_RELATED* and those in red are tainted with *PATCH_RELATED*.

TABLE I. ORIGINAL DEFINITION TRACKING RULES.

Statement	Action[1]	Rule

$r :=^{\ell} \mathrm{R}(r_a)$	if $(S[\sigma[r_a]] == nil)$
	$S[r] = \langle \ell, cnt[\ell] \rangle$;
	else $S[r] = S[\sigma[r_a]]$
		OD-READ

$\mathtt{W}^{\ell}(r_a, r_v)$	$S[\sigma[r_a]] = S[r_v]$	OD-WRITE
$r_d := r_s$	$S[r_d] = S[r_s]$;	OD-COPY
$r :=^{\ell} c$	$S[r] = \langle \ell, cnt[\ell] \rangle$	OD-CONST
$r :=^{\ell} r_1 \text{ op } r_2$	$S[r] = \langle \ell, cnt[\ell] \rangle$	OD-BINOP

TABLE II. PATCHING AND TRACING RULES.

Statement	Action[1]	Rule
$r :=^{\ell} \mathrm{R}(r_a)$	if $(P[\langle \ell, cnt[\ell] \rangle] \neq nil \wedge$ $\tau[\sigma[r_a]] \neq PATCH\_RELATED)$ $\sigma[r] = P[\langle \ell, cnt[\ell] \rangle]$; $T = T \cdot \langle \ell, cnt[\ell], \sigma[r], \tau[r] \rangle$ $cnt[\ell \mapsto (cnt[\ell] + 1)]$	PT-READ
$\mathtt{W}^{\ell}(r_a, r_v)$	$T = T \cdot \langle \ell, cnt[\ell], \sigma[r_v], \tau[r_v] \rangle$ $cnt[\ell \mapsto (cnt[\ell] + 1)]$	PV-WRITE
if (r^{ℓ}) then goto(ℓ_1)	if $(\tau[r] \equiv INPUT\_RELATED)$ { $\sigma[r] =$ select from $true$ or $false$ $T = T \cdot \langle \ell, cnt[\ell], \sigma[r], \tau[r] \rangle$ $cnt[\ell \mapsto (cnt[\ell] + 1)]$	PT-PREDICATE

$r :=^{\ell} c$	if $(P[\langle \ell, cnt[\ell] \rangle] \neq nil)$
	if $(P[\langle \ell, cnt[\ell] \rangle] \neq c)$
	BACKTRACK()
	$T = T \cdot \langle \ell, cnt[\ell], \sigma[r], \tau[r] \rangle$
	$cnt[\ell \mapsto (cnt[\ell] + 1)]$
		PT-CONSTANT

$r :=^{\ell} r_1 \text{ op } r_2$	if $(P[\langle \ell, cnt[\ell] \rangle] \neq nil \wedge$ $\tau[r] \neq PATCH\_RELATED)$ $\sigma[r] = P[\langle \ell, cnt[\ell] \rangle]$; $T = T \cdot \langle \ell, cnt[\ell], \sigma[r], \tau[r] \rangle$ $cnt[\ell \mapsto (cnt[\ell] + 1)]$	PT-BINOP

string "Yes" is loaded and the re-execution becomes consistent.

B. Semantic Rules

The algorithm of P2C in Section IV-A requires executing the program iteratively. Such executions have additional semantics including taint propagation, original definition tracking, patching and tracing. While tainting is standard and hence omitted, we will discuss the later three semantics in this subsection. Note that all such additional semantics are active during each execution in P2C.

Original Definition Tracking. In a symmetric read, when a value is read from the file to an address a, P2C needs to decide where the value is patched to in the original execution. It is undesirable to directly patch to the address. Because in practice, file output operations often occur inside some library functions and hence the address a is often an internal buffer address. Therefore, P2C performs original definition tracking to identify where the value in a was originally defined. The loaded value will be patched at the original definition site. We have shown such an example [C1] in Section IV-A.

The original definition tracking semantic rules are described in Table I. The first column shows the statement and the second column shows the corresponding instrumentation semantics. The name of the rule is shown in the third column. For a memory read (OD-READ), if the value was defined even before P2C starts (i.e. earlier than the file creation point), the current statement instance becomes the original definition of the value. Later, if P2C determines the value should be patched, this is the site where the patch is applied. If the value has a non-empty original definition, P2C simply propagates the definition to register r. Both memory write (OD-WRITE) and register move (OD-COPY) do not change the value, P2C simply copies the original definition. For a binary operation (OD-BINOP), since we cannot say the resulting value in r comes from either r_1 or r_2 in general, its definition site is hence the current statement instance.

Patching and Tracing. During execution, P2C constantly checks if there is a patch for the current statement instance. It also generates an execution trace which will be compared with the trace from the previous execution to detect patch inconsistency, which indicates the presence of unexposed field correlations.

Table II presents the semantic rules for patching and tracing. For a memory read (PT-READ), if there is a patch for the current statement instance and *the current value in the memory was not previously computed from another patch*, the patch is applied. If the latter condition is true, there is patch correlation and the current patch is ignored.

Algorithm 2 Validate Patch Consistency.

```
    Input: $T_1$: the trace with the previous patch;
           $T_2$: the trace with the latest patch ;
 1: function CHECKCONSISTENCY($T_1$,$T_2$)
 2:   if $T_1 \equiv nil \vee T_2 \equiv nil$ then
 3:     return $true$;
 4:   end if
 5:   $\langle \ell_1, i_1, c_1, t_1 \rangle \leftarrow$ POP($T_1$)
 6:   $\langle \ell_2, i_2, c_2, t_2 \rangle \leftarrow$ POP($T_2$)
 7:   if $\ell_1 \equiv \ell_2 \wedge i_1 \equiv i_2 \wedge c_1 \equiv c_2$ then
    /*Perfect match*/
 8:     return CHECKCONSISTENCY($T_1$,$T_2$)
 9:   end if
10:   if $\ell_1 \equiv \ell_2 \wedge i_1 \equiv i_2$ then
    /*Instructions match, but values not*/
11:     if $c_1 \neq c_2 \wedge t_1 \equiv PATCH\_RELATED$ then
    /*Inconsistent patches*/
12:       return $false$
13:     end if
14:     if $c_1 \neq c_2 \wedge t_1 \neq PATCH\_RELATED$ then
    /*Consistent patches*/
15:       if ISPREDICATE($\ell_1$) then
16:         $T_1 \leftarrow$ POPUNITL($T_1$, IMMPOSTDOMINATOR($\ell_1$))
17:         $T_2 \leftarrow$ POPUNITL($T_2$, IMMPOSTDOMINATOR($\ell_1$))
18:       end if
19:       return CHECKCONSISTENCY($T_1$,$T_2$)
20:     end if
21:   end if
22: end function
```

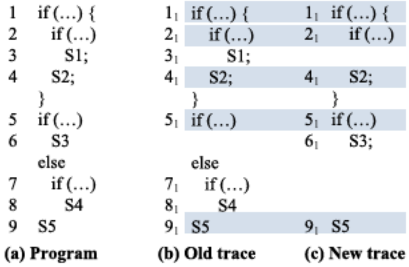
Example. Consider the trace in Fig.7 (c). At Ⓕ, $r0$ is computed from the patched `buf` value and hence has the value 2 and the taint of *PATCH_RELATED*. As such, the current patch is ignored. This leads to inconsistency when compared to trace (b), in which $r0$ is 3. P2C backtracks to Ⓓ and changes the number of bytes to read to 3. This time, `buf` is patched with "`Yes`". As a result, $r0$ has the value 3 and the taint of *PATCH_RELATED* at Ⓕ. As such, even though the patch $P[\langle 4, 1 \rangle]$ is ignored, the traces are consistent. □

The last two actions of the rule PT-READ are to add an entry to the trace and update the statement instance counter. The two actions are standard for all statements. For a predicate statement (PT-PREDICATE), P2C tests if the branch outcome is computed from a random input. If so, the branch outcome should not be trusted. It creates a backtrack point and selects a branch outcome to proceed. Note that later P2C may backtrack and take a different branch. This allows us to handle cases similar to Fig. 3 (d). For a constant assignment (PT-CONSTANT), if there is a patch and the patch is inconsistent with the constant, P2C backtracks. Rule PT-BINOP is similar to PT-READ, P2C checks the taint of the resulting value to determine if the current patch should be applied.

Note the rules presented in this subsection are not the comprehensive set, but rather an important subset. The rules for other statements can be similarly derived.

C. Patch Consistency Validation

P2C progressively transforms individual file output operations to the symmetric input operations. Upon each symmetric read, it creates a new patch entry and then re-executes from the starting point (i.e. the file creation operation) to validate if the new patch entry is consistent with the previously admitted patch entries. The consistency check is by comparing the execution traces with and without the latest patch entry. The criterion is that a trace entry affected by patch must have the same value across runs. In other words, if a trace entry has value computed from some patch entry in one run, it should not have a different value when a new patch entry is applied. We have presented an example of such inconsistency in Section IV-B.



```
1  if (…) {          1_1  if (…) {        1_1  if (…) {
2    if (…)          2_1    if (…)        2_1    if (…)
3      S1;           3_1      S1;
4    S2;             4_1    S2;           4_1    S2;
   }                      }                    }
5  if (…)            5_1  if (…)          5_1  if (…)
6    S3                                   6_1    S3;
   else                   else
7    if (…)          7_1    if (…)
8      S4            8_1      S4
9  S5                9_1  S5              9_1  S5
```

Fig. 8. Trace comparison example. The highlighted entries are those having their values compared.

In practice, consistency checking is more complex than the examples we have presented. The application of patches may cause control flow differences. Sometimes, such differences do not suggest patch inconsistency. The consistency checking algorithm is presented in Algorithm 2. It is a recursive algorithm, which recursively compares the heads of the two given traces until inconsistency is detected or the end of either trace is reached.

Lines 5-6 acquire the head entries of the two traces. If they are identical, the algorithm recursively compares the remaining trace entries (lines 7-9). If the two trace entries have the same statement instances but different values (lines 10-20), there are two possible cases. In the first case (lines 11-13), the entry from the old trace is patch related. Hence, the different value from the new trace entry suggests patch inconsistency. In the second case (lines 14-19), the old entry is not patch related, which suggests the difference is solely caused by the newly introduced patch, not by patch inconsistency. Therefore, the algorithm proceeds to compare the remainders of the two traces.

Special cares need to be taken if the entry is a predicate (line 15) because the value differences indicate different branches will be taken. The algorithm by-passes the control flow differences by discarding trace entries until the immediate post-dominator (i.e. the merge point of the two branches) is reached. As such, the control flows of the two traces are aligned again. In P2C, immediate post-dominators are computed from the control flow graph generated by X-Force.

Example. Fig. 8 shows an example for trace comparison. The program is shown in (a). Figures (b) and (c) present the old and new traces, respectively. The new trace is generated by re-execution after a new patch entry is generated. The traces are simplified, containing only control flow information in order to simplify the discussion. The comparison starts from the head, both statement instances 1_1's have the same branch outcome. The algorithm proceeds to compare 2_1's, which have different branch outcomes. If 2_1 in the old trace is tainted as patch related, inconsistency is detected and the algorithm will return *false*. Assume it is not the case, the algorithm proceeds to the immediate post-dominator of statement 2 in both traces, which is line 4. In other words, 3_1 is discarded from the old trace. Note that it is safe to discard such control flow differences as they are already captured by the comparison of the branch outcomes at 2_1. Similarly, different branch outcomes are detected at 5_1, the algorithm discards 6_1 from the old trace and 7_1, 8_1 from the new trace despite the internal structure of 7_1 because the immediate post-dominator of line 5 is line 9. Observe that the algorithm always aligns the trace entries to ensure comparison is performed at appropriate places. □

The algorithm handles trace entries from loops as at runtime a loop is essentially unrolled to multiple levels of nesting if-statements. We also handle the additional challenges in trace comparison caused by recursive functions even though the details are elided from the algorithm for brevity.

D. Reverse Engineering Field Semantics

We have explained how P2C parses a given file to memory through execution transformation. Note that the successful parsing of the file indicates that the syntactic structure of the file is disclosed. Particularly, the individual fields of the file are recognized by P2C. However, to understand the meaning of the file, we still need to associate semantics to those fields.

In P2C, we leverage the data-flow tracking based type reverse engineering capabilities in X-Force to resolve the field semantics. Specifically, X-Force detects if a memory location has data flow to some API with a known interface. If so, the parameter types and semantics of the interface can be used to type the memory location. Note that P2C patches the field values loaded from the subject file to the execution, which allows us to type these field values by observing the data flow between the patched site and some known APIs. For example in Fig. 2 (b), P2C replaces the original `accounts[0]={` “x#^”,2014,...} with `accounts[0]={` “kjohnson”,2004,...} by the symmetric read at line 52’. Through the data flow from 26 to 52 in the original execution, P2C can type the newly loaded field value `2004` as a year value. X-Force’s type reverse engineering system has a rich set of semantic tags besides the primitive types, such as IP, EMAIL, PASSWORD, and STDIN (denoting that a value originates from stdin). It allows a field to have multiple semantic tags. Users can also add to a pre-defined dictionary that maps API functions to semantic tags.

V. Discussion

Asymmetric Producer and Consumer. So far we have assumed that the producer and the corresponding consumer have symmetric structure such that the sequence of writes in the producer corresponds to the sequence of reads in the consumer. However in practice, the implementation of a real world consumer may not be precisely symmetric to the producer. We have seen cases in which the consumer skips certain data fields when parsing the file, especially when such fields are not critical. In this case, P2C has the unique advantage of providing a better (transformed) consumer that parses all fields. There may be more complex asymmetric cases. Later in Section VI, we will show a case in which the producer first writes a dummy header as a place holder, then writes the main body. After that, it uses `fseek()` to go back and replace the header. Note that this producer may be asymmetric to a regular consumer that reads the header first and then the body. P2C can handle this case because it can generate a consumer that parses the given file by performing the symmetric operations including the symmetric file seek, although the generated consumer may be different from the real world consumer.

Encryption. P2C has limited support in handling encrypted files. In extremal cases, the entire files are encrypted. Specifically, data are usually first stored in a buffer, which is then encrypted and emitted through a single file write. In this case, P2C can only perform the symmetric read and load the entire ciphertext to the buffer. It cannot understand the structure of the fields in the plain-text and their meanings, because achieving this goal is as difficult as decrypting the ciphertext. If the producer can generate different kinds of encrypted files (in different places in the program), P2C will not be able to determine which kind of encrypted file a given subject file belongs to. This is because the given file can be correctly parsed by the (single) symmetric read corresponding to any of the (single) encrypted file writes.

It is common that a data file has part of its data encrypted (e.g. a plaintext header and an encrypted payload). In this case, P2C can still parse and type the plaintext part of the data. For the encrypted part, although P2C cannot recover the corresponding plaintext and its structure, it can still associate types and semantic tags to the entire buffer. For example, it may be able to tell that the encrypted buffer contains user ids and passwords. This is because although the encrypted buffer loses the syntactic structure of its plaintext version, it inherits all the types and semantic tags.

VI. Evaluation

P2C is implemented on PIN [15]. We implement both bit and byte level taint tracking mechanisms. All cases except `Steganography` use a byte level taint tracking due to the overhead. We evaluate P2C on 9 different programs shown in Table III. The first two columns show program names and sizes of program binaries. The third column shows sizes of unknown files and network messages. The next column presents the number of iterations P2C takes to transform the producer execution at the correct output site (to get the desired consumer execution), and the average number of iterations it spends on transforming at a wrong file output site. Here an iteration means a re-execution. Note that a program may output multiple files and only one of the sites is the correct one. The fifth column shows a total execution time to cover both the correct and wrong output sites. The sixth and seventh columns are coverage and the number of paths explored by X-force. The coverage data include both the number and the percentage of instructions covered.

We use 5 programs that write files. `InfoZip` is a file compressor utility. We use an arbitrary zip file created by `InfoZip` to understand the zip file header. `Steganography` is a program that can hide a piece of secret text to an existing bitmap file. We use P2C to understand and extract the secret text from a bitmap produced by the program. `FreePiano` can record and play piano songs. We use P2C to understand a song file format. `Mp3gain` scans and analyzes mp3 files and inserts meta data using ID3 tag. Similarly, `yamdi` inserts meta data into flash video files. We use P2C to understand the meta data format and make sure that they do not add sensitive information.

We also evaluate P2C on 4 network programs to analyze network messages. We use P2C to understand an unknown packet generated by the zbot malware. WinPing sends ICMP messages and PowerOn broadcasts magic packets to turn on computers connected on the same network. NetworkMorris is a network based chess game that also supports chatting between players.

We have a few observations from the results. (1) P2C can correctly parse all the given files/messages. The number of iterations needed to find the appropriate transformation is not large, indicating that field correlations are exposed by program dependences in most cases such that backtracking is not needed. (2) A program may output multiple files and hence have multiple output sites. Only one of them is the one we are looking for. When testing a wrong site, we find that P2C only iterates a very small number of times. In many cases, it terminates in the first execution. That is because different files/messages often have different magic numbers, which allow P2C to detect inconsistency quickly. This also applies to the scenario where it tests a wrong program which produces different file formats. (3) While P2C can precisely identify all the fields in the given files/messages, it can type (or associate semantic tags) to a large portion of the identified fields. In some cases such as Steganography, zbot and FreePiano, all or most fields are correctly typed. The semantic tags have rich semantic information, which will be illustrated by our later case studies. For some programs, they have fields that are not related to any API calls such that P2C cannot type them. For instance, many fields in NetworkMorris denote game board information that is not related to any standard APIs. WinPing also has some fields that are solely arithmetic operation related. yamdi has a lot of floating point fields. But the underlying typing engine of P2C only considers non-floating-point APIs. InfoZip has some fields that are arithmetic and bit operation related such as CRC32. Since we only enable the bit-level taint analysis on Steganography, they are not typed.

Although P2C can associate primitive types to all these fields, we consider them untyped as the information is not helpful enough. On the other hand, since these fields are not related to any APIs (i.e. system calls and library calls), we argue that they may be less important. (4) For the fields that are typed, we manually verify that the derived types (semantic tags) are always correct. (5) We find that some commercial MP3 tag reader programs such as mp3tag are not able to read the injected data by mp3gain whereas our transformed consumer can recognize all of them. (6) X-Force is able to achieve good coverage for most programs except FreePiano and WinPing. FreePiano is a UI intensive program and X-Force currently does not have good support for this kind of programs. In all cases, X-Force is able to cover the correct output sites. (7) The execution time of X-Force is at the scale of a few hours. The execution time for P2C is relatively much smaller, 18 minutes on average. Note that one iteration of the execution takes 40 seconds on average.

A. Case Studies

Next, we will present more details for a few cases. **Stenography.** This program hides a secret string in a bitmap file. Specifically, it stores each bit of the secret data into the least significant bit of a pixel. As the least significant bit of pixel does not exhibit visible differences to human, it is hence difficult to notice the presence of the secret. In this case study, we have a bitmap file which contains an unknown secret text message. Fig. 9 shows how a message "SECRET" is added to a given BMP file. The original bitmap file is on the top, the secret is on the bottom, and the mutated bitmap is in the middle. Note that given a secret message, a string containing the size of the message and the message itself are added to the bitmap file. The program always uses 5 bytes to represent the ASCII version of the size. So the maximum length of the secret message is 99999. The first 55 bytes of the file denote the bitmap file header. Its internal structure is elided for brevity.

Given a file containing an unknown secret, we want to use P2C to recognize where and what the secret is. P2C starts executing the program without the plaintext bitmap file or the secret message. It successfully parses the header (i.e. the first 55 bytes from the secret file) by transforming the header write operations in the original execution that emit a sequence of meaningless random values. These loaded bytes are typed as INPUT_FILE_1, indicating they are related to the first input file, which is supposed to be the plaintext bitmap file.

When the program emits the 56th byte, P2C performs the symmetric read of the corresponding byte and identifies that the original definitions of the first 7 bits are from a file read but the remaining 1 bit is from a result buffer of a call to the sprintf(...,"%d",...) API[5]. P2C patches the original bits accordingly. In fact, the following 39 bytes from the secret file have a bit that originates from the same buffer. After the symmetric reads of these bytes, P2C successfully reconstructs the buffer that holds the length of the secret, which is 6. This length value is later used to guide the program execution to read the least bit from the following $6 * 8 = 48$ bytes. The original definition of those bits is a return buffer of a file read operation in the original execution, with the type of INPUT_FILE_2, indicating they are from the second input file, which is supposed to contain the secret message. By patching the buffer, the original secret can be recovered by P2C. Note that P2C does not require monitoring the original execution that generated the secret file. It does not even require the user to provide valid inputs to run the program. Compared to techniques that reverse engineer output format [11], [4] and hence aim to construct a general grammar for all bitmap files with secret embedded, which may be difficult, P2C leverages the producer logic to understand a given specific file.

InfoZip We use an arbitrary zip file to evaluate P2C on the InfoZip program. Handling InfoZip is challenging as it extensively uses file seek operations. Specifically, when the program writes a zip file, it first writes a dummy file header then appends the compressed content. The dummy file header does not have a correct checksum value yet. After it finishes the compression, then it seeks to the dummy file header and rewrites the file header with the correct values. Fig. 10 (a) shows a substantially simplified code snippet. It stores a value writ and uses it in file seek later. Column (b) shows the execution in which the symmetric read at line 3 is performed. Column (c) shows the execution in which zfi.siz is patched but the value of writ is still problematic.

[5] Note that our original definition tracking can be configured to be performed at the bit level.

TABLE III. EVALUATION RESULTS

Program	Program Size	Unknown File Size	# of iterations (Matched / Unmatched)	# of typed fields / # of fields	Elapsed Time	Coverage / # of total inst.	# of paths explored
InfoZip	37 KB	5 KB	12 / 0[1]	19 / 35	2m 45s	6015 / 6015	385
Steganography	17 KB	4219 KB	24 / 0[1]	3 / 3	9m 27s	631 / 974	12
FreePiano	2296 KB	6 KB	42 / 0[1]	6 / 6	28m 35s	47991 / 286571	2531
mp3gain[†]	109 KB	1304 KB	17 / 0[1]	6 / 10	24m 42s	21672 / 21672	754
yamdi	225 KB	102 KB	38 / 0[1]	34 / 72	1h 16m 38s	21491 / 24577	1183
zbot	26 KB	30 Bytes	7 / 2	5 / 7	8m 48s	3089 / 3089	199
WinPing	244 KB	45 Bytes	5 / 0[1]	4 / 8	2m 27s	14758 / 29620	983
PowerOn	25 KB	102 Bytes	1 / 0[1]	2 / 2	21s	709 / 1058	29
NetworkMorris	1049 KB	52 Bytes	20 / 4	15 / 39	8m 6s	3057 / 7814	186

[1] It means P2C terminates in the initial execution without backtracking.

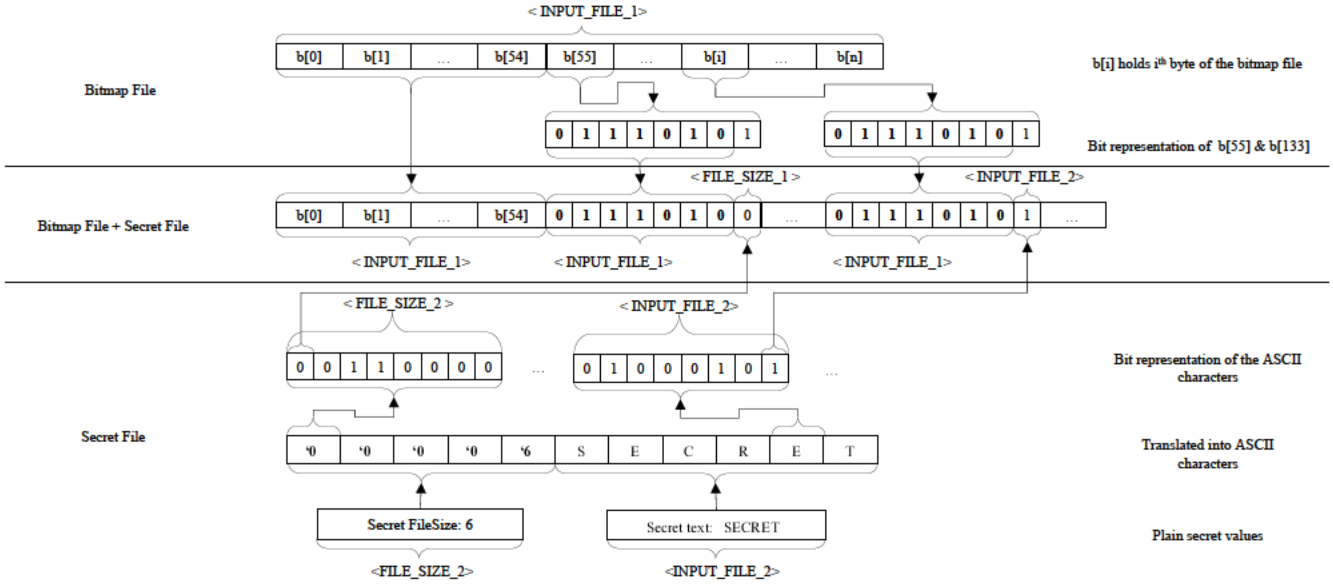


Fig. 9. Data flow and type information of the bitmap steganography program. FILE_SIZE_2 denotes that the value represents the size of second input file (i.e. the one containing the secrete message) and INPUT_FILE_i represents the value is related to the *i*th input file.

```
    const sig = 0x04034b50;
     [...]
1:  writ += csize;
2:  zfi siz = csize+passex;
3:  write(&zfi.siz);
     [...]
4:  seek(writ);
     [...]
5:  write(sig);
```

(a)

```
    [...] writ == 0
1:  writ, csize == 100
2:  zfi.siz == 100
3:  read(&zfi.siz) == 110
```

(b)

```
    [...] writ == 0
1:  writ, csize == 100
2:  zfi.siz == 110
3:  read(&zfi.siz) == 110
     [...]
4:  writ == 100
     [...]
5:  read(sig) == 0x313632
```

(c)

Fig. 10. `InfoZip`. Code in (a); traces in (b) and (c). Particularly, the patch on `zfi.siz` is applied in (c). The file operations are highlighted.

In the execution in Fig. 10 (b), `zfi.siz` has a value 100 because `csize` and `passex` are 100 and 0 respectively. Then, P2C reads the corresponding value of `zfi.siz` from the unknown file, which is 110, and generates a patch for line 2. In the next run (c), although the value of `zfi.siz` is corrected (110), `writ` still has a wrong value (100), leading to a wrong file seek operation at line 5. This is because we cannot patch `csize`: the correct value of `csize` cannot be inferred from the correct value of `zfi.siz`. Fortunately, P2C detects the parameter to file seek (at line 4) is uncertain. It creates a backtrack point and proceeds with the current value 100. The following file write (line 5) is to write a magic number as part of the header. When P2C performs the symmetric read, since the previous file seek got to a wrong starting point of the header region in the unknown file, the loaded value is inconsistent with the constant magic number. As such, P2C backtracks and tries a different value of `writ`. Note that it cannot go beyond line 5 until the file seek operation becomes correct. In this case, P2C iterates 12 times. After that, the producer execution is correctly transformed to parse the zip file.

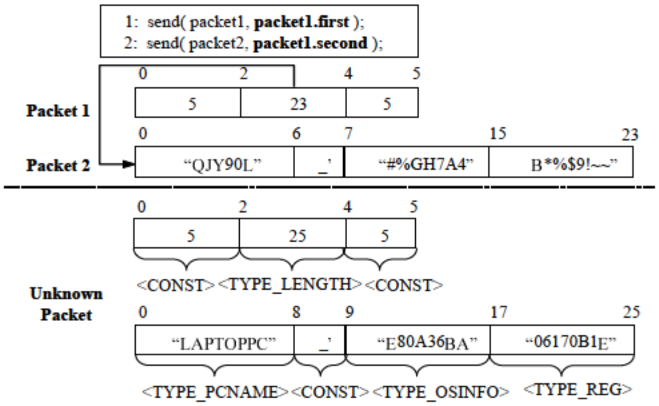
zbot. `zbot` is trojan malware that communicates via networks. We apply P2C to the `zbot` client program in order to understand some packets that we generated in a different execution, which simulate a previously generated, unknown message. In practice, such messages could come from a network monitoring system that logs network messages that are unusual. We find that the `zbot` client can create different types of network messages, depending on the execution paths. Therefore, this may create difficulty for dynamic analysis based output format reverse engineering technique [4], [20] that can generate output message grammars. In particular, the user must drive the malware execution to cover the path that generates the intended grammar.

In contrast, given a message, P2C is able to get to the producers of different messages and test one by one to see which one is the producer for the given message.

In this case, a valid message contains two packets [6]. The first packet contains some constant magic number that is

[6] Since the packets are sent consecutively inside a method, P2C applies transformation to the body of the method.

different for different kinds of messages. It also contains the size for the second packet.



```
1: send( packet1, packet1.first );
2: send( packet2, packet1.second );
```

Fig. 11. Zbot case. The unknown message is on the bottom and the corresponding meaningless message by the producer is on the top.

Fig. 11 shows the unknown message and the corresponding meaningless message. In the first iteration of P2C, it reaches the bulk write of the packet 1, it performs the symmetric read, which loads the first packet from the unknown message. Note that the first field of packet 1, denoted as `packet1.first`, is a magic number. This allows us to detect inconsistency if the current producer is not for the unknown message as the magic number would be different. The second field denotes the length of the second packet. The symmetric read correctly patches the length variable in the execution. The third field of packet 1 is supposed to be a command, which is also correctly patched.

P2C proceeds to the second packet send and performs the symmetric read that loads the second packet in the unknown message. When it tries to patch the execution with the loaded values, it identifies that the original definitions of the first 6 bytes come from a buffer definition inside a call `sprintf(s,"%s_%8X_%8X",...)` with a variable length, which suggests the length is uncertain. In other words, we do not know how many bytes we should copy from the loaded second packet back to the buffer. P2C automatically generates a backtrack point and proceeds with the current value of 6. As a result, the first 6 bytes “LAPTOP” are copied to the buffer. Furthermore, the original definition of the 7th byte corresponds to the constant definition of letter ‘_’. As such, P2C copies the 7th byte from the second packet, which is ‘P’ to the definition point, causing inconsistency. P2C backtracks and tries to copy a longer sequence to the first field until the correct string “LAPTOPPC” is patched. Once the first field is correctly parsed, the remaining two fields can be easily parsed as they have fixed length.

The type tags of the fields in the meaningless message (the top one in Fig. 11) are passed onto the corresponding fields in the unknown message. As such, we recognize that the first field denotes <TYPE_PCNAME> as it gets its value from `GetComputerName()`. The following two fields are from `GetOsVersion()` and `RegQueryValue()`.

VII. Related Work

Protocol/Input-file format reverse engineering. There have been a large body of work on protocol and input-file format reverse engineering [5], [12], [21], [6], [16], [4], [20], [13]. These techniques are mostly dynamic analysis based. They monitor program execution to detect format disclosing instructions and track the data-flow between these instructions and input file/message fields through taint analysis. For example, a memory read in a loop accessing consecutive bytes inside a given packet/file indicates a buffer field in the file/message [5], [4]; comparison instructions often indicate hierarchy between fields [6], [13]. In particular, some of them can reverse engineer out-going message format and handle encrypted messages [4], [20]. Some of them can generate message grammars [21], [6], [16]. These techniques require monitoring the program execution that parses or generates the messages/files, which implies the users have to know how to run the program and provide a few concrete inputs. In our application scenario, we only assume the user knows the producer binary. He does not need to know how to run the program or provide inputs. In fact, we can easily extend P2C such that it can search through a list of potential producers to find the right producer. More importantly, given a subject unknown file/message, *P2C does not assume the original execution that produced the file/message was monitored* such that those techniques cannot be directly used to derive the format and the meanings. In addition, these techniques are dynamic analysis based such that the results depend on the inputs provided. As such, the generated grammar may be incomplete. In contrast, P2C does not try to generate a general grammar, but rather parse and understand a specific file by transforming an arbitrary producer execution to the exact consumer execution that parses the file/message. Coverage is hence not a concern for P2C. Consequently, P2C has a different set of technical challenges compared to these techniques. It focuses on generating the proper consumer execution. After that, it leverages similar taint-analysis based techniques to type fields.

Static Analysis based Output File Format Reverse Engineering. Lim et al. proposed using static binary analysis to analyze producer programs to construct regular expressions that can be used to parse output files [11]. However, the technique over-approximates the grammar of the output file due to the conservativeness of the underlying static analysis. It cannot express field correlations. As such, a buffer field with variant length will be denoted by a kleene closure in the resulting regular expression, which would accept many other strings. Hence, although the regular expression can parse a file, the generated parse tree may not reflect the real structure. Furthermore, the technique does not associate meaning to individual fields.

In [8], Driscoll et al. proposed a static technique that checks the conformance of producer and consumer. It reports any non-conformances as potential bugs. Although it has a different goal, the observation that producers and consumers are symmetric inspired P2C.

Binary Type Reverse Engineering. Techniques [14], [10], [19], [4] have been proposed to reverse engineer data structure definitions (e.g., field types) from binaries. They can also reverse engineer semantic information to a certain extent such as timestamps and IP addresses. These techniques work by observing data flow between variables and API functions with known interfaces. P2C leverages a similar technique to type data fields after they are parsed to memory. Note that these techniques cannot be used to solve our problem as in our case, the subject file/message was generated in the past and there is no way to track that producer execution. The contribution of P2C lies in reconstructing a transformed execution that parses the file such that types can be recovered by monitoring the transformed execution. Balakrishnan et al. [1], [2], [18] have shown that by statically analyzing executables alone can largely discover syntactic structure of variables, such as sizes, field offsets, and simple structures. Their technique entails points-to analysis and abstract interpretation at binary level, which may be difficult to scale to large and complex binaries. They cannot be directly used to solve our problem.

Binary Reuse. P2C is also related to binary code reuse [3], [9], [7] as it leverages the existing producer logic to derive the consumer execution. BCR [3] identifies and reuses encryption/decryption functions in malware using both static and dynamic analysis. Inspector Gadget[9] leverages dynamic slicing for identifying specific malware behavior for extraction, reuse, and analysis. Virtuoso[7] applies dynamic slicing to identify program logic in applications running in a VM, which can then be reused for VM introspection. Compared to these techniques, P2C does not extract any components. It tries to leverage the producer logic in the context of the producer execution and performs on the fly transformation.

VIII. Conclusion

We develop P2C, a system that can understand the structure and meaning of an unknown file or network message that was generated in the past, without the availability of their consumer. It explores a set of potential producers to find out the real one. It iteratively and progressively transforms a producer execution to a consumer execution that closely resembles the ideal consumer execution that can parse and process the file/message, leveraging the symmetry between producer and consumer. It does not require the user to know the exact producer, where in the producer the file/message was generated, or how to run the producer. It also features the capability of handling unexposed field correlations. Experiments on a set of real world programs show that the technique is highly effective.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments. This research was supported, in part, by DARPA under Cooperative Agreement HR0011-12-2-0006, NSF under Award 1409668, and a gift from Cisco Systems. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] G. Balakrishnan, , G. Balakrishnan, and T. Reps. Analyzing memory accesses in x86 executables. In *Proc. of Intenational Conference on Compiler Construction (CC)*, pages 5–23. Springer-Verlag, 2004.

[2] G. Balakrishnan and T. Reps. Divine: Discovering variables in executables. In *Proc. of Internation Conf. on Verification Model Checking and Abstract Interpretation (VMCAI)*, Nice, France, 2007. ACM Press.

[3] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *Proc. 17th Annual Network and Distributed System Security Symposium*, 2010.

[4] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proc. of the 16th ACM Conference on Computer and and Communications Security (CCS'09)*, 2009.

[5] J. Caballero and D. Song. Polyglot: Automatic extraction of protocol format using dynamic binary analysis. In *Proc. of the 14th ACM Conference on Computer and and Communications Security (CCS'07)*, 2007.

[6] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proc. of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, Alexandria, VA, October 2008.

[7] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proc. 2011 IEEE Symposium on Security and Privacy*, 2011.

[8] E. Driscoll, A. Burton, and T. Reps. Checking conformance of a producer and a consumer. In *Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*, 2011.

[9] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proc. 2010 IEEE Symposium on Security and Privacy*, 2010.

[10] J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. In *Proc. 18th Annual Network and Distributed System Security Symposium*, 2011.

[11] J. Lim, T. Reps, and B. Liblit. Extracting output formats from executables. In *Proc. of Working Conference on Reverse Engineering (WCRE)*, 2006.

[12] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proc. of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, San Diego, CA, February 2008.

[13] Z. Lin and X. Zhang. Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2008.

[14] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proc. 17th Annual Network and Distributed System Security Symposium*, 2010.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, 2005.

[16] P. Milani Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol Specification Extraction. In *IEEE Symposium on Security & Privacy*, 2009.

[17] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su. X-force: Force execution binaries for security applications. In *Proc. USENIX Security Symposium*, 2014.

[18] T. W. Reps and G. Balakrishnan. Improved memory-access analysis for x86 executables. In *Proc. of Intenational Conference on Compiler Construction (CC)*, pages 16–35, 2008.

[19] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proc. 18th Annual Network and Distributed System Security Symposium*, 2011.

[20] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Proc. of 14th European Symposium on Research in Computer Security (ESORICS'09)*, Saint Malo, France, September 2009. LNCS.

[21] G. Wondracek, P. Milani, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Proc. of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, San Diego, CA, February 2008.