Extracting Output Formats from Executables*

Junghee Lim
junghee@cs.wisc.edu

Thomas Reps
reps@cs.wisc.edu

Ben Liblit
liblit@cs.wisc.edu

Computer Sciences Department, University of Wisconsin-Madison†

Abstract

We describe the design and implementation of FFE/x86 (File-Format Extractor for x86), an analysis tool that works on stripped executables (i.e., neither source code nor debugging information need be available) and extracts output data formats, such as file formats and network packet formats. We first construct a Hierarchical Finite State Machine (HFSM) that over-approximates the output data format. An HFSM defines a language over the operations used to generate output data. We use Value-Set Analysis (VSA) and Aggregate Structure Identification (ASI) to annotate HFSMs with information that partially characterizes some of the output data values. VSA determines an over-approximation of the set of addresses and integer values that each data object can hold at each program point, and ASI analyzes memory accesses in the program to recover information about the structure of aggregates. A series of filtering operations is performed to over-approximate an HFSM with a finite-state machine, which can result in a final answer that is easier to understand. Our experiments with FFE/x86 uncovered a possible bug in the image-conversion utility `png2ico`.

1. Introduction

Reverse engineering helps one gain insight into a program's internal workings. It is often performed to retrieve the source code of a program (e.g., because the source code was lost), to analyze a program that may be malicious (such as a virus), to fix a bug, to improve the performance of a program, and so forth. This paper describes a reverse-engineering tool that can help a human understand what a program produces as its output.

As COTS (Commercial Off-The-Shelf) software is increasingly deployed (for which source code and documentation of proprietary intermediate formats are often not available), reverse engineering becomes increasingly needed for interoperability. When a COTS tool uses a proprietary file format, interoperability can be inhibited: the tool can only be used in a tool chain with a consumer or producer of files that have that format.

The technique presented in this paper promotes the reuse of components of a tool chain. For example, when a software engineer wants to build a program that can process the files that a COTS software product generates, he can use our tool to obtain information about the format specification, which would be useful when creating a program that can act as a substitute consumer (or producer).

The technique presented here might also be useful in malware detection. For instance, when trying to identify live versions of the same malware, one would like to have a way to figure out the format of its network traffic. Our technique can provide help with this problem.

Furthermore, our technique can provide a summary of a program's behavior: it produces a structure that consists of a reduced number of entities (compared with the call graph for instance), which may make it easier to understand what the program is doing.

The contributions of our work are:

- It provides a technique for extracting an over-approximation of a program's output data format, including
  - a way to extract a preliminary structure for the output data format (§3)
  - a way to elaborate the structure by annotating it with information about possible output values and sizes (§4)
  - a way to simplify the structure to provide greater understanding of the output data format (§5)

  This provides information that can lead to greater understanding of a program's behavior.
- We report experimental results from applying *FFE/x86* on three applications. Our experiments uncovered a possible bug in `png2ico` (see §7.2).

Although we have concentrated on the problem of extracting output file formats from executables, the same approach could be applied to source code (where one could also take advantage of information about the program's

*This work was supported in part by NSF under grants CCF-0524051 and CCR-9986308, and by ONR under contracts N00014-01-1-{0796,0708}.

†1210 W. Dayton St., Madison, WI 53706, USA.

variables and their declared types), as well as to extracting input file formats.

The remainder of the paper is organized as follows: §2 discusses the key observations that inspired our work and the assumptions for our approach. §3 explains the process of constructing a structure for the output data format, and also provides an overview of the infrastructure on which our implementation is based. §4 discusses how to elaborate the structure generated from the first step with static analyses. §5 presents a series of filtering operations for making HFSMs more understandable. §6 describes how we validated *FFE/x86*. §7 presents experimental results. §8 describes related work. §9 describes possible future directions.

2. Observations & assumptions

2.1. Programming styles

This section makes a few observations about programming styles used in typical application programs to produce output data.

Programming styles relevant to writing output data can be categorized as *individual writes* and *bulk writes*. We present different approaches tailored to handle them in later sections. (Some programs use both styles; our tool is capable of handling such programs, as well.)

```
[1] void put_byte(char c)
[2] {...}
[3] void put_long(long c)
[4] {...}
[5] void writes(char* c)
[6] {...}
[7] void type() {
[8]  switch(...)  {
[9]  case 0:
[10]  put_byte('a');
[11]  break;
[12] case 1:
[13]  put_byte('b');
[14]  break;
[15] }
[16]}
[17]void chksum() {
[18] put_long(...);
[19]}
[20]void fill_data() {
[21] while(...)  {
[22]  put_byte(c);
[23]  ...
[24] }
[25]}
[26]void main() {
[27] put_long(magic1)
[28] put_long(magic2)
[29] writes(filename);
[30] type();
[31] put_long(size);
[32] chksum();
[33] return 0;
[34]}
```

Figure 1. An example that uses individual writes.

Individual writes. The first programming style is to write individual data items out separately to a file or a network. Standard I/O functions, such as *fputs* and *fputc* in C programs, could be used. In practice, however, *wrapper functions* tend to be frequently used. Fig. 1 shows an example of this programming style using wrapper functions, such as `put_byte`, `put_long`, and `writes`. Several fields of the output, including magic numbers, types, sizes, and a checksum, are written out by calling wrapper functions. These functions provide an API to append output items to an internal buffer; once the whole buffer has been filled, the contents of the buffer are flushed. Whereas the buffer is written out in bulk, the individual calls to the wrapper functions represent the "individual writes" referred to in our name for this style. We refer to both the standard I/O functions and user-defined wrapper functions as *output functions*.

An *output operation* is an operation relevant to generating an output data object. Specifically, the term output operation is defined as a call site that calls an output function—either a standard I/O library function or a wrapper function (see lines 10, 13, 18, 22, 27, 28, 29, and 31 in Fig. 1).

Our experience so far is that many application programs are coded in this programming style. For instance, `gzip`, [6][1] `compress95` [2], and `png2ico` [8] follow such a programming style.

```
[1] typedef struct header {
[2]  byte magic[2];
[3]  char name[100];
[4]  char type;
[5]  long size;
[6]  long chksum;
[7] } header;
[8] void write_file() {
[9]  header* h;
[10] h=(header*)malloc(...);
[11] h->magic[0] = ...;
[12] strcpy(h->name, ...);
[13] h->type = ...;
[14] h->size = ...;
[15] h->chksum = ...;
[16] fwrite(h,
[17]   sizeof(header), 1,fp);
[18] write_data();
[19] ...
[20]}
```

Figure 2. An example of a bulk write.

Bulk writes. The second programming style is to use `struct`s or classes to manipulate headers. Fig. 2 shows an example of using a header structure to write output data. A header `struct` object is created at line 10. Each field of the `struct` is set to some value in lines 11–15. Finally, at lines 16–17, the object is written out to the file in its entirety. In this programming style, calls like the one to `fwrite` are the output operations.

In practice, we observed that `tar` [9] and `cpio` [3] use such aggregate structures as storage in preparation for a bulk write. We suspect that this style would be used for more than just headers by applications whose output files consist of a sequence of records.

2.2. User-supplied information

In our current implementation, the user must identify the output functions and supply some additional information about them, in particular, information about each output-relevant parameter:

- whether it is a numeric value to be written out

[1] Because the `gzip` source uses macros instead of functions, output operations are not call sites in the `gzip` executable. This is not compatible with our approach of having the user identify the output operations by supplying the names of output functions. To convert `gzip` into an example in which output operations are visible as procedure calls—so that it could be used for proof of concept in our experimental study—we modified the `gzip` source code to change all output macro definitions into explicit functions. Automatically identifying low-level code fragments that represent output operations remains a challenging problem for future work.

- whether it is an address pointing to the memory containing the data to be written out
- whether it indicates how many bytes are written out

See §4.1 for more details. In the case of standard I/O functions, such information is already known.

3. First step

In our approach, a *Hierarchical Finite State Machine (HFSM)* is used to represent an output data format. An HFSM is a structure in which nesting of finite automata within states is allowed [12, 13]. An HFSM captures commonalities by organizing states in such a hierarchy. Note the following two points about HFSMs:

- The languages of paths in recursive HFSMs are exactly the context-free languages.
- The languages of paths in non-recursive HFSMs are the regular languages.

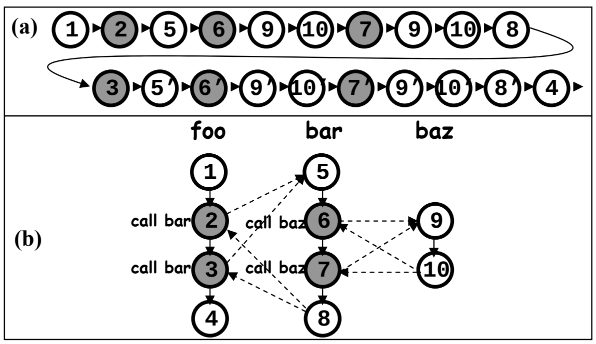


Figure 3. (a) An FSM, (b) A hierarchical FSM.

However, non-recursive hierarchical FSMs can be exponentially more succinct than conventional FSMs due to sharing, as illustrated in Fig. 3.

3.1. Construction of an HFSM

We will use the code fragment shown in Fig. 1 to explain our approach. The code emulates an archive utility. It writes two magic numbers, followed by the file's name, layout type, size, and check-sum, using wrapper functions. Fig. 5 shows its disassembled code as generated by IDAPro.

Each procedure involved with at least one output operation gives rise to an FSM. The program's wrapper functions include `put_byte` (`sub_401050` in the disassembled code), `put_long` (`sub_401075`), and `writes` (`sub_4010E4`), and calls to these functions represent output operations. *FFE/x86* finds the output operations and constructs an HFSM based on the CFGs provided by CodeSurfer/x86 [23]. Our analyzer creates a reduced interprocedural control-flow graph (i.e., the HFSM) that is the projection of the interprocedural control-flow graph onto enter nodes, exit nodes, call nodes, and output operations.

Fig. 4 shows the outcome from running *FFE/x86*. Each node in the HFSM is either an output operation (such as

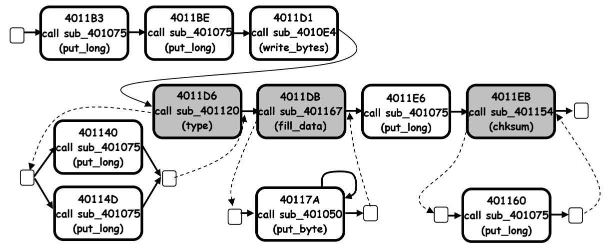


Figure 4. The HFSM for Fig. 1. The shaded boxes signify calls to FSMs. Dotted lines indicate implicit connections between FSMs.

```
401120 sub_401120 proc near; type
401120      push    ebp
401121      mov     ebp, esp
401123      sub     esp, 0Ch
401126      mov     eax, [ebp-4]
401129      mov     [ebp-8], eax
40112C      cmp     [ebp-8], 0
401130      jz      short loc_40113A
401132      cmp     [ebp-8], 1
401136      jz      short loc_401147
401138      jmp     short loc_401152
40113A loc_40113A:
40113A      mov     eax, [ebp-4]
40113D      mov     [esp], eax
401140      call    sub_401050
401145      jmp     short loc_401152
401147 loc_401147:
401147      mov     eax, [ebp-4]
40114A      mov     [esp], eax
40114D      call    sub_401050
401152 loc_401152:
401152      leave
401153      retn
401154 sub_401154 proc near; chksum
401154      push    ebp
401155      mov     ebp, esp
401157      sub     esp, 8
40115A      mov     eax, [ebp-4]
40115D      mov     [esp], eax
401160      call    sub_401075
401165      leave
401166      retn
401167 sub_401167 proc near; fill_data
401167      push    ebp
401168      mov     ebp, esp
40116A      sub     esp, 8
40116D loc_40116D:
40116D      cmp     [ebp-1], 0
401171      jz      short loc_401181
401173      movsx   eax, [ebp-1]
401177      mov     [esp], eax
40117A      call    sub_401050
40117F      jmp     short loc_40116D
401181 loc_401181:
401181      leave
401182      retn
```

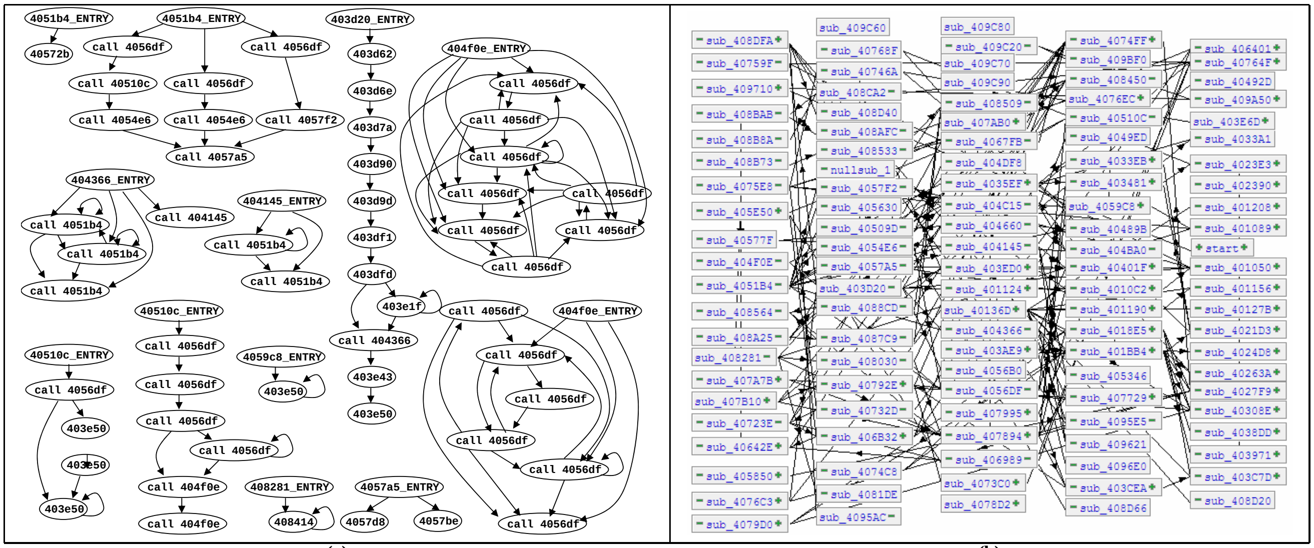
```
401183 sub_401183 proc near; main
401183      push    ebp
401184      mov     ebp, esp
401186      sub     esp, 28h
401189      and     esp, 0FFFFFFF0h
40118C      mov     eax, 0
401191      add     eax, 0Fh
401194      add     eax, 0Fh
401197      shr     eax, 4
40119A      shl     eax, 4
40119D      mov     [ebp-14h], eax
4011A0      mov     eax, [ebp-14h]
4011A3      call    sub_401200
4011A8      call    __main
4011AD      mov     eax, [ebp-10h]
4011B0      mov     [esp], eax
4011B3      call    sub_401075
4011B8      mov     eax, [ebp-0Ch]
4011BB      mov     [esp], eax
4011BE      call    sub_401075
4011C3      mov     [esp+4], 4
4011CB      mov     eax, [ebp-8]
4011CE      mov     [esp], eax
4011D1      call    sub_4010E4
4011D6      call    sub_401120
4011DB      call    sub_401167
4011E0      mov     eax, [ebp-4]
4011E3      mov     [esp], eax
4011E6      call    sub_401075
4011EB      call    sub_401154
4011F0      mov     eax, 0
4011F5      leave
4011F6      retn
```

Figure 5. The disassembled code for Fig. 1. Transparent boxes indicate output operations, and shaded boxes indicate calls to sub-FSMs.

`4011B3`) or a call-site (such as `4011D6`) to a sub-FSM (such as `type`). A call-site node, which represents a call to a sub-FSM, implicitly connects the two FSMs in the HFSM.

The HFSM generated by our tool for `gzip` is shown in Fig. 6(a). Our thesis is that HFSMs (including elaborations and refinements of HFSMs, as explained in §4 and §5) provide a basis for gaining an understanding of the program's behavior. In this regard, it is instructive to compare the HFSM with the program's call graph, because a call graph is another structure that a programmer may use to gain a high-level understanding of a program.

Fig. 6(b) shows a part of the call graph for `gzip`. `Gzip` is composed of 114 control-flow graphs (CFGs), 11491 CFG nodes, and 625 call sites. Even though the HFSM produced by our tool appears to be quite complicated, it is substantially less complicated than both the program's call graph and its interprocedural control-flow graph: the HFSM for `gzip` has 12 FSMs, 64 nodes, and 36 call sites.



(a) (b)

Figure 6. (a) The HFSM for `gzip`. (b) a fragment of the call graph of `gzip`.

3.2. Existing infrastructure

FFE/x86 uses intermediate representations (IRs) provided by the CodeSurfer/x86 framework (Fig. 7), which provides an analyst with a powerful and flexible platform for investigating the properties and behaviors of x86 executables [23]. CodeSurfer/x86 includes several static analyses, including *Value Set Analysis (VSA)* [14, 24] and *Aggregate Structure Identification (ASI)* [15].

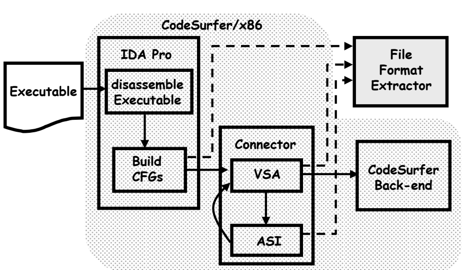


Figure 7. Organization of *CoderSurfer/x86*, and how *FFE/x86* interacts with its components.

VSA is a combined numeric-analysis and pointer-analysis algorithm that determines an over-approximation of the set of numeric values and addresses that each memory location holds at each program point [14]. ASI recovers information about variables and types, especially for aggregates, including arrays and structs. The variables recovered by ASI are used by VSA to obtain information about the variables' possible values. The values recovered by VSA are used by ASI to identify a refined set of variables. Thus, CodeSurfer/x86 runs VSA and ASI repeatedly, either until quiescence, or until some user-supplied bound is reached.[2]

CodeSurfer/x86 uses an initial estimate of the program's variables, the call graph, and control-flow graphs (CFGs) for the program's procedures provided by IDAPro. IDAPro itself does not identify the targets of all indirect jumps and indirect calls, and therefore the call graph and control-flow graphs that it constructs are not complete. In contrast, CodeSurfer/x86 uses the values that VSA discovers to resolve indirect jumps and indirect calls, and thus is able to supply a sound over-approximation to the call graph.

§4 discusses other ways in which VSA and ASI can be exploited for our purposes.

4. Augmenting an HFSM with static-analyses information

In this section, we explain how to exploit the static analyses mentioned in §3.2 for elaborating HFSMs.

4.1. Value Set Analysis

The HFSM generated by the method described in §3.1 provides some information for understanding an output format. The HFSM can be made more precise by annotating it with additional information. In particular, we wish to label each node with information about:

- the size (in bytes) of the data that the node represents
- an over-approximation of the value written out

The values of interest are the actual parameters corresponding to the formal parameters of output functions. For example, suppose that `put_byte` is one of the output functions (see Fig. 8(a)). Suppose that at one of the call sites that

[2]If VSA and ASI have not quiesced when the bound is reached, it is still safe to use the results from the final round of VSA. In particular, each round of VSA provides an over-approximation of the set of numeric values and addresses for each memory location, modulo the treatment of possible memory-safety violations—some of which may be due to loss of precision during VSA. See [14] for more details.

```
void put_byte(char c) {
  outbuf[outcnt++] = (uch)(c);
  if(outcnt==OUTBUFSIZE)
    flush_outbuf();
}
```
(a)

```
mov   byte ptr[esp], 1Fh
call  put_byte
```
(b)

Figure 8. An example code fragment; `put_byte` is a output function, and call sites that call it are output operations.

calls `put_byte` (i.e., at one of the output operations), the actual parameter is always 1Fh (see Fig. 8(b)). This information can be obtained from the information collected by VSA. Note that at the call on `put_byte`, the relevant value is stored on the stack in the byte pointed to by `esp`. The abstract memory configuration (AMC) that VSA would have for the call site would indicate this: for instance, Fig. 9(a) illustrates the values that the AMC would contain in this example. In particular, our tool is able to obtain an over-approximation of the set of values that the actual may hold by evaluating the operand expression `[esp]` in the AMC, which amounts to looking up in the AMC the contents of the cell (or cells) that `esp` may point to. (For this example, the result would be a singleton set, namely, $\{$1Fh$\}$.)

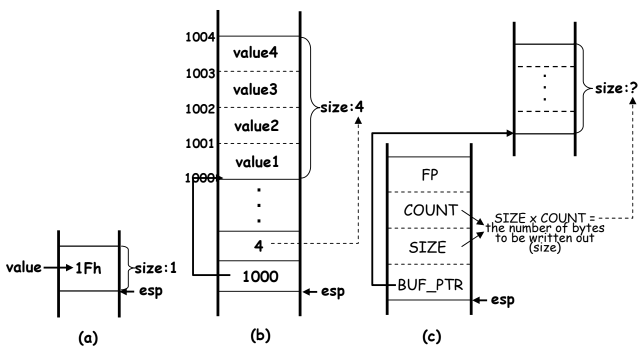


Figure 9. How to obtain information from VSA.

There are two kinds of parameters that can be passed into a output function: numeric values and addresses.

Numeric values. The case where an actual parameter holds a numeric value has been already explained above (see Fig. 9(a)). The corresponding size of the value can be obtained from ASI, which infers the size from the usage pattern of the formal parameter in the called function. (In the case where an output operation calls a standard I/O function, this information is available from the signature of the function.) For example, `put_byte` would have a 1-byte argument, `put_short` a 2-byte argument, and so forth.

Addresses. If the type of a formal parameter is a pointer, the set of addresses in the memory location corresponding to the actual parameter would be used to look up in the AMC the values in the cells to which the actual parameter could point (see Fig. 9(b)).

The case of `fwrite` at lines 16–17 in Fig. 2 falls into this category. The address of the heap-allocated memory location that contains the data is passed as the first argument.

`size_t fwrite(const void` **BUF_PTR*, `size_t` *SIZE*, `size_t` *COUNT*, `FILE` **FP*);

It is known that the product of the second and third parameters of `fwrite` is the number of bytes that are written out (see Fig. 9(c)).

Value roles. The kind of abstract value recovered by VSA sometimes suggests what the value's role is, e.g.,

- `Singleton` - If VSA recovers a singleton value for an actual parameter of an output operation, the parameter may correspond to either a magic number or a reserved field.
- `Set of numeric values` - If the value that VSA recovers is a non-singleton set of numeric values, the parameter may correspond to an optional field.
- `Top` - If VSA gives *Top*, which means any value, for an actual parameter of an output operation, the parameter may correspond to variant data.

4.2. Aggregate Structure Identification

As mentioned in §2, programmers frequently use a `struct` or a class to collect data before it is written out.

```
[1] u_char outpack[MAXPACKET];
[2] static void pinger(void) {
[3]   register struct icmphdr *icp;
[4]   register int cc;
[5]   int i;
[6]   icp = (struct icmphdr*)outpack;
[7]   icp->icmp_type = ICMP_ECHO;
[8]   icp->icmp_code = 0;
[9]   icp->icmp_cksum = 0;
[10]  icp->icmp_seq = ntransmitted++;
[11]  icp->icmp_id = ident;
[12]  ...
[13]  i = sendto(s, (char*)outpack, cc, 0, &whereto,
[14]            sizeof(struct sockaddr));
[15]  ...
[16]}
```

Figure 10. Code fragment used to illustrate the use of ASI information.

Fig. 10 shows a fragment from `ping` [7] in which a network packet is constructed. Instead of writing individual data items one at a time using output operations, a `struct` object is used to store output data while multiple fields are prepared, as shown in lines 7–11 of Fig. 10. Then the aggregate object is written out (i.e., sent out) all together on lines 13–14.

Aggregate Structure Identification (ASI) [15, 22] is a unification-based, flow-insensitive algorithm to identify the structure of aggregates in a program. Whenever a read or write to a part of a memory object is encountered, ASI records how the memory object should be subdivided into smaller objects that are consistent with the memory access.

In this example, we assume that the user has indicated that `sendto`, which is a GNU C library function, is the

```
[1] mov eax, dword ptr [ebp - 10h]
[2] mov byte ptr [eax], 8
[3] mov edx, dword ptr [ebp - 10h]
[4] mov byte ptr [edx + 1], 0
[5] mov eax, dword ptr [ebp - 10h]
[6] mov word ptr [eax + 2], 0
[7] mov eax, dword ptr [ntransmitted]
[8] mov edx, dword ptr [ebp - 10h]
[9] mov word ptr [edx + 6], ax
[10]inc dword ptr [ntransmitted]
[11]mov eax, dword ptr [ident]
[12]mov edx, dword ptr [ebp - 10h]
[13]mov word ptr [edx + 4], ax
```

(a)

```
Global:
 struct {
   ...
   byte_1 outpack.0;
   byte_1 outpack.1;
   byte_2 outpack.2;
   byte_2 outpack.4;
   byte_2 outpack.6;
   ...
 }
```

(b)

Figure 11. (a) The disassembled code fragment for Fig. 10, (b) The outcome of ASI.

only output function. The second argument of `sendto` is known to be a pointer to a `struct` object with unknown substructure. ASI provides information about this substructure. The instructions that correspond to the assignment statements at lines 7–11 of Fig. 10 are shown in Fig. 11(a) at lines 2, 4, 6, 9, and 13, respectively. VSA provides information about the extent of memory accessed by each of these instructions. ASI uses that information to subdivide the portion of memory accessed, thereby producing the structure shown in Fig. 11(b). This indicates that the structure of the packet header may consist of two 1-byte fields, followed by three 2-byte fields.

ASI is also capable of recovering information about the structure of aggregates that are allocated in the heap.

This example illustrates a case where each output function emits a completely-constructed chunk of output data, and the HFSM represents the program's output operations at a high level of abstraction. In bulk writes as this example, structure information recovered by ASI can help identifying the structure of output data format. This can be seen in Fig. 17(b), where `pinger`'s call to `sendto` is elaborated as a sequence of 1- and 2-byte header-field writes, followed by a larger packet payload.

5. Filtering

Because an HFSM can be hard to understand, we experimented with applying a series of filtering operations—including simplification, conversion of each FSM to a regular expression, and inline expansion—to generate a simpler representation of the output format as a regular expression. In our experiments, this has been done manually; however, the process would be relatively easy to automate.

Simplification. Not all nodes in the HFSM are helpful in understanding an output format. An unnecessarily complicated HFSM could prevent users from understanding key aspects of an output format.

Most portions of the HFSM shown in Fig. 6(a) turn out to be either `Top-value`, `Top-size`, or an unbounded loop that includes them. `Top-value` means that the node could have any value; `Top-size` means that the node could be of any size.

In each of the following cases, a node (or a node set) would not provide meaningful information:

- A node of `Top-size` and `Top-value`
- A node set in an unbounded loop, each of which has both `Top-size` and `Top-value`

To be considered as a *meaningful node*, a node must be

- A node of non-`Top-size`

Algorithm 1 Simplification algorithm.

Input: HFSM
Output: Trimed HFSM
  Set the status of all FSMs to be *meaningful*
  while There exists a *meaningful* FSM that contains only *non-meaningful nodes* or calls to *non-meaningful FSMs* **do**
    Set M to be a *non-meaningful FSM*
    Transform M into an FSM with a self-loop on a node labeled with (`Top-size/Top-value`)
  end while

Alg. 1 describes an algorithm for simplifying HFSMs generated by *FFE/x86*. The idea behind the algorithm is to consider the cases mentioned above: for an FSM that consists of only nodes with `Top-value` and `Top-size`, or an unbounded loop that includes only such items, it may be better to simplify it to $(Top)^*$ because the original FSM would not provide much meaningful information about the output format.

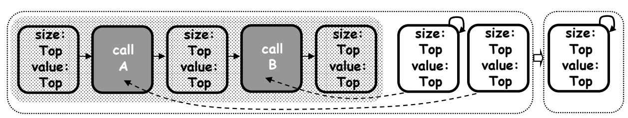


Figure 12. An example of simplification.

Fig. 12 shows an example of simplification. The shaded FSM that contains two *non-meaningful FSM*s and three *non-meaningful node*s is simplified to an unbounded self-loop consisting of a node (`Top-size`/`Top-value`).

Conversion to a regular expression. We can convert each FSM in an HFSM into a regular expression using the Kleene construction.

Expansion. The final step is to apply inline expansion. Recursion was not encountered in any of the applications that we used for our experiments (see §7), so inline expansion could be applied without worrying about non-termination. If recursion had been encountered, we could have summarized strongly connected components of the call graph.

Fig. 13 represents the final outcome from using these techniques on our example.

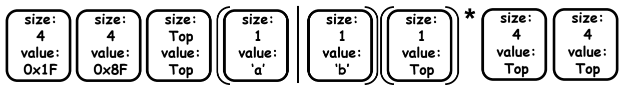


Figure 13. The final result after simplification, conversion, and inline expansion.

6. Validation against dynamic output

We validate our approach by testing whether the outcome from our algorithm (i.e., the regular expression) matches output data produced during actual runs of the application.

We used *flex* [5], a tool for generating scanners for compilers. Given an input specification in the form of a list of pattern-action pairs (where the pattern is a regular expression), *flex* generates a program that repeatedly finds the longest prefix of the (remaining) input that matches one of the patterns. To create a tool for testing whether a regular expression R generated by our algorithm describes the output of an application, we give *flex* a 2-pattern specification—consisting of R (with an action to report success), plus a default pattern (with an action to report failure).

As discussed earlier, each box (as shown in Fig. 13) in the regular expression generated by our technique is labeled with two kinds of information: a value and a size. Value and size are either `Top`, a `Singleton`, or a set of numeric values.

- `Singleton`
- `A set of numeric values`
- `Top`

Thus, to be able to feed it to *flex*, the regular expression needs to be transformed to one in which the basic unit is a 1-byte character. Table 1 shows the transformation rules that are applied to boxes.[3]

Table 1. Transformation of boxes.

size	value	conversion
`Singleton` n	`Singleton`	According to the value of *n*, this is split into multiple boxes that contain a 1-byte value. (E.g., the first box in Fig. 14(a) is transformed to the first four boxes in Fig. 14(b).)
`Singleton` n	`Top`	`Top` is transformed to '.', which matches any character. Thus, this is transformed to a sequence of n boxes that contain '.'. (E.g., the fifth box in Fig. 14(a) is transformed to the last two boxes in Fig. 14(b).)
`Top`	`Top`	This is transformed to a box that contains '.' with a self-loop. (E.g., the third box in Fig. 14 (a) is transformed to the box that has a loop in Fig. 14(b).)

Table 1 describes only the cases when size and value have either `Singleton` or `Top`.(Note that there is no case when size is `Top` and the value is non-`Top` because this is not a possible outcome of VSA.) For the case when either size, value, or both have a set of numeric values, we split

[3] We use '.' as a shorthand for "any character". In *flex*, it is necessary to use the pattern '.|\n'.

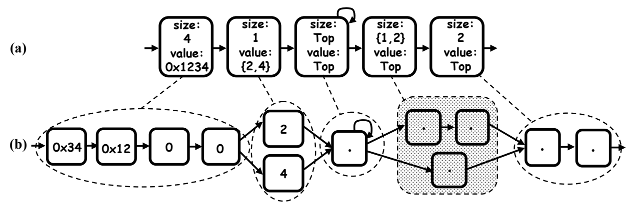


Figure 14. An example of the transformation. '.' means any character.

the box into multiple boxes that have a `Singleton` value and a `Singleton` size. For example, the second box in Fig. 14(a), which has two values (2 and 4), is transformed to the two boxes in Fig. 14(b) that have the values 2 and 4, respectively. For the case where size is not a `Singleton`, the shaded boxes in Fig. 14(b) show how it is converted.

Note that this process is only for validation, because the original values or sets of values are more likely to be understandable to a human than the subdivided values.

7. Experimental results

We evaluated *FFE/x86* on three applications: `gzip`, `png2ico`, and `ping`.

7.1. `gzip`

`Gzip` is a GNU data-compression program. Fig. 15 represents the outcome after filtering the HFSM from Fig. 6(a).

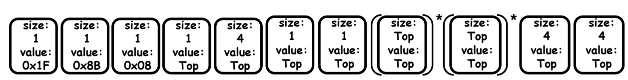
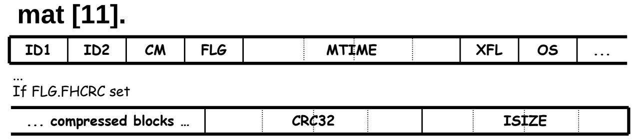


Figure 15. The final result for `gzip`.

Table 2. Part of the specification of `gzip`'s format [11].



ID1 and ID2	These are the fixed values: ID1=31 (0xlF), ID2=139 (0x8B)
CM	This identifies compression method: CM=0-7 are reserved, CM=8 denotes the "deflate" compression method.
FLG	This is divided into individual bits: bit 0 FTEXT, bit 1 FHCRC and so forth.
MTIME	This gives the most recent modification time of the original file being compressed.
XFL	This is available for use by specific compression methods.
OS	This identifies the type of file system on which compression took place: 0 - FAT filesystem, 1 - Amiga, and so forth.
CRC32	This contains a cyclic redundancy check value of the uncompressed data.
ISIZE	This contains the size of the original input data modulo 2^{32}.

The format of `.gz` files generated by `gzip` is described in RFC 1952 (see Table 2). The outcome shown in Fig. 15 correctly over-approximates the specification. In other words, the language of the outcome is a superset of the output language of `gzip`. The outcome has the two magic numbers (`ID1`=0x1f and `ID2`=0x8b) and a constant

(CM=8) at the same positions shown in Table 2. This is followed by a 4-byte element (corresponding to MTIME), two 1-byte elements (corresponding to XFL and OS). At the end, it has two 4-byte elements, which correspond to CRC32 and ISIZE.

We also applied the validation process described in §6 to this outcome. The *flex*-generated validator accepted each of five .gz files (chosen arbitrarily from the Internet).

7.2. png2ico

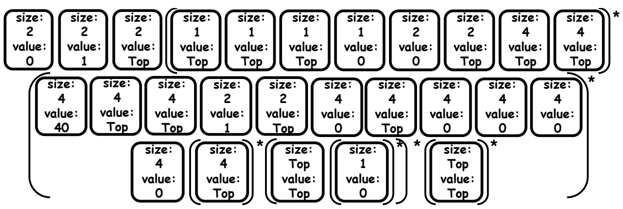


Figure 16. The outcome for png2ico.

Table 3. An unofficial specification of the ico format [1].

Name	Size	Description
Reserved	2 byte	=0
Type	2 byte	=1
Count	2 byte	Number of Icons in this file
Entries	Count * 16	List of icons
Width	1 byte	Cursor Width (16, 32 or 64)
Height	1 byte	Cursor Height (16, 32 or 64 , most commonly = Width)
ColorCount	1 byte	Number of Colors (2,16, 0=256)
Reserved	1 byte	=0
Planes	**2 byte**	**=1**
BitCount	2 byte	bits per pixel (1, 4, 8)
SizeInBytes	4 byte	Size of (InfoHeader + ANDbitmap + XORbitmap)
FileOffset	4 byte	FilePos, where InfoHeader starts
repeated Count times		
InfoHeader	40 bytes	Variant of BMP infoHeader
Size	4 bytes	Size of InfoHeader structure = 40
Width	4 bytes	Icon Width
Height	4 bytes	Icon Height (added height of XOR-Bitmap and AND-Bitmap)
Planes	2 bytes	number of planes = 1
BitCount	2 bytes	bits per pixel = 1, 4, 8
Compression	4 bytes	Type of Compression = 0
ImageSize	4 bytes	Size of Image in Bytes = 0 (uncompressed)
XpixelsPerM	4 bytes	unused = 0
YpixelsPerM	4 bytes	unused = 0
ColorsUsed	4 bytes	unused = 0
ColorsImportant	4 bytes	unused = 0
Colors	Number-of-Colors * 4 bytes	Color Map for XOR-Bitmap
Red	1 byte	red component
Green	1 byte	green component
Blue	1 byte	blue component
reserved	1 byte	=0
repeated NumberOfColors times		
XORBitmap	...	bitmap
ANDBitmap	...	monochrome bitmap

Png2ico converts PNG files to Windows icon-resource files. Fig. 16 shows the final outcome. Compared with an unofficial specification of the ico image format [1] given in Table 3, most of the constant data items in the format have been recovered by *FFE/x86*. For example, several fields in the ico format, including Reserved and Type, have constant values that are recovered through our technique. Furthermore, the overall structure of Fig. 16 is similar to Table 3. One difference is that the format recovered by *FFE/x86* shows two loops at top level: one for a sequence of Entries, and one for a sequence of structures that each consist of an InfoHeader, a sequence of Colors, a sequence of XORBitmaps, and a sequence of ANDBitmaps. In constrast, Table 3 shows only a single InfoHeader/Color/XORBitmapANDBitmap structure. An inspection of the source code confirmed that png2ico definitely supports a sequence of InfoHeader/Color/XORBitmapANDBitmap structures.

FFE/x86 also revealed a possible bug in png2ico—that is, it showed that the format produced by png2ico does not satisfy the specification given in Table 3. According to Table 3, the Planes field of Entries should be 1; however, as shown by the eighth box with size=2 and value=0 in the first row of Fig. 16, png2ico always produces 0, rather than 1. This discrepency was discovered when we ran the *flex*-generated validator (which checks for conformance to the png2ico output format extracted by *FFE/x86*) on some pre-existing .ico files. Those files came from a Windows XP installation (and presumably were not created by running the freeware png2ico utility). The validator rejected those files, but accepted all 23 .ico files that we generated using png2ico. We tracked down the problem to the following line in the png2ico source:

```
writeWord(outfile,0); //wPlanes
```

7.3. ping

Ping [7] sends ICMP ECHO_REQUEST packets to a host to see if the host is reachable via the network. Sendto is the only output function of ping.

As discussed in §4.2, the whole structure of the HFSM shown in Fig. 17(a) represents the program's output operations at a high level of abstraction. From the HFSM, it can be inferred that main calls pinger and catcher, and pinger calls sendto. The pinger sub-FSM (see Fig. 17(b)), which is constructed from the information recovered for sendto by the ASI, has a format where the sizes of successive elements are 1, 1, 2, 2, and 2 bytes, respectively, as shown in Fig. 11(a).

As shown in Fig. 18, the icmp packet struct includes two 1-byte fields (uint8 icmp_type and uint8 icmp_code), one 2-byte field (uint16 icmp_checksum), and two unions—icmp_hun and icmp_dun. The outcome from *FFE/x86* satisfies a part of the specification. The first two 1-byte fields match with uint8 icmp_type and uint8 icmp_code,

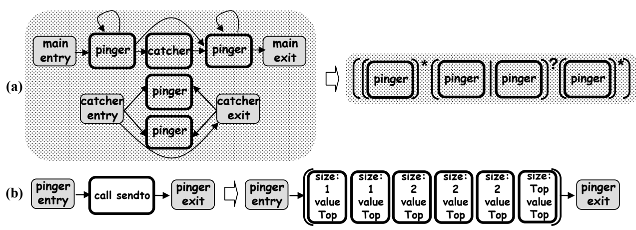


Figure 17. The outcome for `ping`**. (a) The HFSM hints the program behavior of** `ping`**, (b) The packet contains an 8-byte** `icmp` **header followed by data.**

```
typedef struct icmp {
        uint8 icmp_type;                        /* type of message, see below */
        uint8 icmp_code;                        /* type sub code */
        uint16 icmp_checksum;                   /* ones complement cksum of struct */
#define icmp_cksum icmp_checksum
        union {
                uint8 ih_pptr;                  /* ICMP_PARAMPROB */
                struct in_addr ih_gwaddr;       /* ICMP_REDIRECT */
                struct ih_idseq {
                        uint16 icd_id;
                        uint16 icd_seq;
                } ih_idseq;
                int ih_void;
                /* ICMP_UNREACH_NEEDFRAG -- Path MTU Discovery (RFC1191) */
                struct ih_pmtu {
                        uint16 ipm_void;
                        uint16 ipm_nextmtu;
                } ih_pmtu;
                struct ih_rtradv {
                        uint8 irt_num_addrs;
                        uint8 irt_wpa;
                        uint16 irt_lifetime;
                } ih_rtradv;
        } icmp_hun;
#define icmp_pptr icmp_hun.ih_pptr
...
        union {
                struct id_ts {
                        uint32 its_otime;
                        uint32 its_rtime;
                        uint32 its_ttime;
                } id_ts;
                struct id_ip {
                        struct ip idi_ip;
                        /* options and then 64 bits of data */
                } id_ip;
                struct icmp_ra_addr id_radv;
                uint32 id_mask;
                char id_data[1];
        } icmp_dun;
#define icmp_otime icmp_dun.id_ts.its_otime
...
} icmp_t;
```

Figure 18. The `icmp` **packet structure [10].**

respectively. The first 2-byte field matches with `uint8 icmp_cksum`. The last two 2-byte fields match with the first union, namely, `icmp_hun`, which includes a `struct ih_idseq` that consists of `uint16 icd_id` and `uint16 icd_seq`.

However, the last union (`icmp_dun`) was not discovered by ASI: there is no assignment to that union in the code, and thus ASI does not partition the memory locations to which the union corresponds.

`Signal`. The outcome from *FFE/x86* is incomplete in one respect: as shown in Fig. 19, lines 1–2, `ping` calls the `signal` library function. `Signal` allows asynchronous event handling, which means that the statically generated control-flow graph might not cover all possible flows of control. Our technique is based on a CFG statically generated by CodeSurfer/x86. Thus, if output operations ap-

```
[1] (void)signal(SITINT, finish);
[2] (void)signal(SIGALRM, catcher);
[3] while(preload--)
[4]   pinger();
[5] if((options & F_FLOOD) == 0)
[6]   catcher(0);
[7] for(;;) {
[8]   struct sockaddr_in from;
[9]   register int cc;
[10]  size_t fromlen;
[11]  if(options & F_FLOOD) {
[12]    if(floodok) {
[13]      floodok = 0;
[14]      pinger();
[15]    }
[16]    ...
[17]  }
[18]  ...
[19]}
```

Figure 19. A code fragment from `ping`**.**

pear in the handler function that a `signal` call establishes, the resultant HFSM might not over-approximate all possible outputs.

8. Related work

Most previous work on reverse engineering of file formats has been dynamic and manual. Eilam describes a strategy for deciphering file formats given a symbol table and a sample output file [19]. This approach requires manually stepping through disassembled code and inspecting memory contents in a debugger while the program produces the given file. Other approaches ignore the program and rely on heuristic generalization from one or more sample output files. For example, one reverse-engineering case study searched for `zlib`-compressed data, file names, length bytes, and other typical structures [4]. All of these approaches require considerable manual effort and one cannot guarantee that the chosen sample files are sufficiently general. In constrast, the static approach described here over-approximates a file format without relying on sample files, symbol tables, or extensive manual analysis. Human intervention is only needed to identify output functions and to assign higher-level interpretations (e.g., "file name") to selected fields identified by the analysis.

There have been similar attempts to statically recover information about program data. Christensen et al. have presented a technique for discovering the possible values of string expressions in Java programs [17]. First, a context-free grammar is generated by constructing dependence graphs from class files. The grammar is then widened into a regular language, which contains all possible strings that could be dynamically generated.

The method of Christensen et al. has also been applied to low-level code; Christodorescu et al. used the method in a string analysis for x86 executables [18]. This approach is similar to ours in the sense that x86 executables are the targets of both tools, and the recovered output data format in the analysis is represented as a regular language that denotes a superset of the actual output language. Their approach, however, is different from ours in the sense that the initial context-free structure recovered by their tool comes from the structure of operations purely internal to each procedure, rather than from the call-return structure of the program, as in our tool.

Our approach is also related to work on host-based intrusion detection, in which models of expected program behavior are also constructed. The model over-approximates the possible sequences of system calls, and, by comparing the actual sequence of system calls to those allowed by the model, is used to detect when malicious input has hijacked the program. Pushdown-system models have been employed for this purpose, either constructed from source code [25] or from low-level code [20, 21] (in particular, SPARC executables). Our HFSMs are similar in that they also yield context-free languages that are a projection of a portion of the program's behavior. We have gone beyond previous work by using the results from two dataflow analyses (namely, VSA and ASI) to elaborate our models with information about possible sets of values and value sizes.

9. Conclusion and future work

In this paper, we focus on output operations. However, the same approach can be applied to other kinds of operations. For example, one could treat *input operations*, which are associated with examining or parsing an input file, using the same approach taken in this paper. In this case, one would want to consider only paths to exit points that represent successful runs of the program (because these correspond to successful uses of an well-formed input files). In addition, one could apply our approach to network communication operations that parse or construct packets.

As suggested by one of the referees, it may be possible to use such a characterization of the input language as a way to generate test inputs. Similarly, knowledge of the output language for component c_1 in a tool chain could be used as a source of test inputs for the next component c_2 in the chain.

As described in the discussion of `ping`, `signal` calls are a factor that can cause the HFSM to not over-approximate the actual output language of the program. The only description of a static-analysis tool that is able to handle such features is the paper on MOPS [16]. The approach used in MOPS could be used with our HFSMs, as well.

As mentioned earlier, we assume that output functions are identified by the user. To create a more automatic tool for extracting data formats, it would be desirable to find a way to automatically identify output functions, especially wrapper functions.

Each loop in an HFSM is currently transformed to either $(\texttt{node-set})^*$ or $(\texttt{node-set})^+$. However, there can be cases when the bound on the number of possible iterations of a loop can be obtained from VSA. In such cases, the information about a loop's iteration bounds would provide users with more precise information about the output format.

References

[1] Basic file format for *ICO* files. "http://www.daubnet.com/formats/ICO.html".

[2] *compress95*, SPEC benchmark. "http://www.itee.uq.edu.au/∼emmerik/specbench.html".

[3] *cpio*, *GNU* project. "http://www.gnu.org/software/cpio/cpio.html".

[4] *File Format Reversing - EverQuest II VPK*. "http://www.openrce.org/articles/full_vew/16".

[5] *flex*. "http://www.gnu.org/software/flex/".

[6] *gzip*, *GNU* project. "http://www.gzip.org/".

[7] *ping*. "http://packages.debian.org/stable/net/netkit-ping".

[8] *png2ico*. "http://www.winterdrache.de/freeware/png2ico/".

[9] *tar*, *GNU* project. "http://www.gnu.org/software/tar/tar.html".

[10] FreeBSD/Linux Kernel Cross Reference. "http://fxr.watson.org/fxr/source/netinet/ip_icmp.h".

[11] *GZIP* file format specification version 4.3. "http://www.gzip.org/zlib/rfc-gzip.html".

[12] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *TOPLAS*, 27(4), 2005.

[13] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *FSE*, pages 175–188, 1998.

[14] G. Balakrishnan. and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, 2004.

[15] G. Balakrishnan and T. Reps. Recovery of variables and heap structure in x86 executables. Tech. Rep. TR-1533, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, Sept. 2005.

[16] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properites of software. In *CCS*, 2002.

[17] A. S. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. In *SAS*, 2003.

[18] M. Christodorescu, N. Kidd, and W. Goh. String analysis for x86 binaries. In *PASTE*, 2005.

[19] E. Eilam. *Reversing—Secrets of Reverse Engineering*. Wiley Publishing, Inc., 2005.

[20] J. T. Giffin, S. Jha, and B. Miller. Detecting manipulated remote call streams. In *USENIX Security Symposium*, 2002.

[21] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *NDSS*, 2004.

[22] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *POPL*, pages 119–132, 1999.

[23] T. Reps., G. Balakrishnan, and J. Lim. A next-generation platform for analyzing executables. In *APLAS*, 2005.

[24] T. Reps., G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *PEPM*, 2006.

[25] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, 2001.