Bin-Carver: Automatic Recovery of Binary Executable Files

By

Scott Hand, Zhiqiang Lin, Guofei Gu
and Bhavani Thuraisingham

From the proceedings of
The Digital Forensic Research Conference
DFRWS 2012 USA
Washington, DC (Aug 6th - 8th)

DFRWS is dedicated to the sharing of knowledge and ideas about digital forensics research. Ever since it organized the first open workshop devoted to digital forensics in 2001, DFRWS continues to bring academics and practitioners together in an informal environment.

As a non-profit, volunteer organization, DFRWS sponsors technical working groups, annual conferences and challenges to help drive the direction of research and development.

http://dfrws.org
Bin-Carver: Automatic recovery of binary executable files

Scott Hand, Zhiqiang Lin, Guofei Gu, Bhavani Thuraisingham

1Department of Computer Science, The University of Texas at Dallas, USA
2Department of Computer Science and Engineering, Texas A&M University, USA

Abstract

File carving is the process of reassembling files from disk fragments based on the file content in the absence of file system metadata. By leveraging both file header and footer pairs, traditional file carving mainly focuses on document and image files such as PDF and JPEG. With the vast amount of malware code appearing in the wild daily, recovery of binary executable files becomes an important problem, especially for the case in which malware deletes itself after compromising a computer. However, unlike image files that usually have both a header and footer pair, executable files only have header information, which makes the carving much harder. In this paper, we present Bin-Carver, a first-of-its-kind system to automatically recover executable files with deleted or corrupted metadata. The key idea is to explore the road map information defined in executable file headers and the explicit control flow paths present in the binary code. Our experiment with thousands of binary code files has shown our Bin-Carver to be incredibly accurate, with an identification rate of 96.3% and recovery rate of 93.1% on average when handling file systems ranging from pristine to chaotic and highly fragmented.

© 2012 Z. Lin, S. Hand, G. Gu & B. Thuraisingham. Published by Elsevier Ltd. All rights reserved.

1. Introduction

File carving (Pal et al., 2003, 2008; Garfinkel, 2007) is a process by which raw data is examined with the goal of reconstructing previously indexed files when file system metadata such as super-blocks, directory entries (dentry), and index-node (inode) tables are corrupted or missing. File carving is feasible because, when most operating systems delete a file, the file's content is not overwritten; only the metadata is affected. This is partly due to the fact that file systems are optimized for performance and protecting their security and privacy is not a primary concern.

File carving has largely been used for data recovery (Pal and Memon, 2009) (a lucrative market recently), such as restoring deleted files or recovering data from a damaged device. The need for this utility stems from the fact that end-users sometimes unintentionally “permanently” delete a file, and sometimes disk directory content is corrupted (by malware for instance). File carving is also useful for digital forensics (Garfinkel, 2007), as it can recover evidence files that have been “deleted” by criminals.

File carving, theoretically a permutation problem, is challenging for several reasons. For example, it is often difficult to determine where files begin and end without some sort of data block index to provide the block order for a single file. Moreover, even though we can detect the start and end of a file, it is still not sufficient because of fragmentation (Garfinkel, 2007), which can cause a contiguous sequence of file blocks to be split into two or more contiguous sequences.

Traditional file carving techniques utilize header-footer pairs to identify file boundaries, leading to the recovery of files that are contiguous on the media (Pal et al., 2003, 2008; Garfinkel, 2007; Pal and Memon, 2009). For example, all JPEG files begin with the hexadecimal sequence FF D8 and end with FF D9 (JPEG file interchange format file format summary). By looking for this kind of unique sequences, this approach is effective for certain types of files...
containing reliable header-footer signatures (such as JPEG files) in the absence of fragmentation. However, it is weak when applied to heavily fragmented files or those without exact header and footer information.

Meanwhile, traditional file carving focuses mainly on files of common forensic interest such as documents (e.g., TXT/DOC/PDF), images (e.g., JPEG/GIF/PNG), audio (e.g., WAV/WMF/MP3), and video (e.g., AVI/MPEG/MP4) files (Garfinkel, 2007; Pal and Memon, 2009). Fewer attempts have been made toward the recovery of binary executables. However, binary executable recovery is also useful. At the very least it can narrow down the search of the traditional carving space. That is, when given a disk image, we can exclude the binary executable files if we can identify them and only focus on other chunks of the disk.

On the other hand, binary executable carving has become an important concern recently to security and forensic investigation. This is because in the past few years, we have witnessed an exponential increase of malicious executable files. For example, according to a report from AV-Test (Year-end malware stats from av-test), they processed an average of 54k malware samples daily in 2010 (up from an average of 33k in 2009, and 426 in 1998). Furthermore, malware code often deletes itself (to remove its footprints) after accomplishing certain tasks. It is therefore helpful to have an approach with which executable files on a disk image can be enumerated, mapped, and recovered.

Thus, in this paper, as a first-in-its-kind proof-of-concept for binary executable file carving, we present a novel system, Bin-Carver, to focus in particular on ELF executables in the Linux/UNIX platform (Note that PE (Microsoft pe and coff specification) files for Windows share the same methodology as ELF files for UNIX and thus can be handled in a similar way). Our system is fully automatic. The key idea is not only that it explores the file structure information from the ELF headers, but also that it explores the intrinsic characteristics of the binary code such as the code distributions and the explicit control flow path present in the code. In particular, inspired by the image file carving which uses magic numbers to identify both headers and footers, we use magic numbers to first identify ELF headers, which will serve as a road map to recover all other sections/segments (Executable and linkable format). Because of fragmentation, the ELF header alone is not sufficient, and we further explore the binary code content. We especially focus on the explicit function call control flow path as a guideline with which to efficiently solve the fragmentation problem.

Bin-Carver focuses on the recovery of the binary code for the widely used x86 architecture. Since we leverage the internal control flow path of the binary code to handle fragmentation, it would appear that we have to solve the disassembly problem, which is challenging because x86 instructions could start at any address (i.e., there is no address alignment constraint). Fortunately, by looking at the code distributions, we find that the code sequence of the function call control flow path has unique signatures. Thus, Bin-Carver will not disassemble all of the binary code; it will only inspect a small amount to construct the control flow.

The main contribution of this paper is summarized as follows:

- We make the first step in recovering fragmented executable files in a disk image.
- We present the use of the road map information defined in executable file headers and the explicit control flow paths contained in the binary code to guide the recovery of the executable file.
- We have implemented a tool called Bin-Carver, and applied it to recover thousands of binary executables from a number of disk images. Our experimental results show that our technique achieves extraordinary accuracy even with large fragmentation.

2. Approach overview

In this section, we first define our file carving problem, then outline the challenges, and finally provide an overview of our system.

2.1. Problem statement and assumptions

In this paper, we aim to recover an ELF executable file e from a disk image D in the presence of only the file content blocks. Meanwhile, we only focus on recovery of the file content and do not attempt to recover its filename. For proof-of-concept and simplicity, we assume a Linux platform with EXT2 file system and block size 4K. It is important to note that, while newer file systems such as EXT3 and EXT4 are more commonly used in Linux machines recently, there is no difference in how the data blocks are laid out. EXT3 and EXT4 add improvements such as journaling and higher-level metadata that helps reduce fragmentation, but core structures remain similar enough that file carving accuracy is the same. EXT2 is used because its relative simplicity is useful in writing tools to evaluate accuracy.

We also assume the file content has not been overwritten. In addition, we assume the file content is stored in an increasing order in the disk. While this may not always be the case, we believe that violations of this assumption are rare enough that we can ignore the possibility for the purpose of evaluating this algorithm. At least in our evaluation with the EXT2 file system, we did not encounter such a case.

More formally, as illustrated in Fig. 1, assume that for an ELF file e that has n blocks in the disk, our goal is to link these n blocks together to eventually recover the files.
such, our solution is a graph-based approach. In particular, we utilize the internal logic between graph nodes to connect the data blocks. Note that the block address $A(b_i)$ is greater than $A(b_j)$ (the increasing order assumption), and there may be zero, one, or more data "garbage" gaps with size 4K each between the data block $b_i$ and $b_j$ where $i, j \in \{0, n - 1\}$.

2.2. Challenges

In the absence of file system metadata such as the inode and dentry, file carving is challenging. For instance, we cannot recover the filename that is managed in the dentry even though the dentry is not corrupted (overwritten). We made a detailed case study in the EXT2 file system (our experimental file system) and found that the inode pointer is cleared in the corresponding dentry. Consequentially, we cannot correlate any dentry with the inode to recover the filename. That explains why most file carving systems cannot recover filenames.

However, the most challenging problem is the fragmentation. A study by Garfinkel in 2007 (Garfinkel, 2007) showed that 15% of binary executables were found to be fragmented. However, in our experiments with EXT2 file systems, we discovered that the incidence of non-contiguous data block sequences is commonly even higher. For example, we took a /bin/ls binary with 92,376 bytes (requiring at least 23 4K data blocks) and observed how this binary file gets organized in a brand new disk. We found that this simple ls file actually gets fragmented (not stored contiguously) in the disk. In particular, we found this file requires 24 data blocks and there is a special data block which stores the pointer to the other data blocks. An investigation with the EXT2 file system data structure reveals that the 13th data block stores the one-layer indirect data block pointers. Therefore, in an EXT2 file system with 4K data blocks, the probability for a file with size larger than 48K being fragmented is very high, because there are only 12 direct data block pointers in the inode.

An intuitive approach that filters the indirect block by looking at the block content may be able to solve this problem. However, there are many other situations that can cause disk fragmentation. For example, on a disk already exhibiting a large degree of fragmentation, storing large files or appending data to existing files may further increase fragmentation, because there may not be a sufficient number of contiguous data blocks to store the new data.

2.3. System overview

An overview of our system is presented in Fig. 2. There are three key components in our system: ELF-header scanner, block-node linker, and conflict-node resolver. As the first step, ELF-header scanner is used to scan all possible ELF headers $h_i$ using the ELF-file magic value. After that, guided by the road map from each $h_i$ and the internal control flow of the binary code, block-node linker scans the disk image and tries to identify all the possible nodes and link them together. Finally, our conflict-node resolver will remove the conflict nodes introduced by our linker due to the fragmentation or possible garbage data to eventually output the ELF-file $e_i$.

3. Bin-Carver design

3.1. ELF-header scanner

The ELF header always resides at the beginning of an ELF file data block, and holds a “road map” describing the organization of an ELF file (Executable and linkable format). By searching for the magic number sequence 7f 45 4c 46 (.ELF) at the starting address of each data block (4K in our case) in $D$, we are able to locate the ELF headers, which contain a wealth of information on how to traverse all other code/data sections.

```
typedef struct {
  00 unsigned char e_ident [16];
  16 Elf32_Half e_type;
  18 Elf32_Half e_machine;
  20 Elf32_Word e_version;
  24 Elf32_Addr e_entry;
  28 Elf32_Off e_phoff;
  32 Elf32_Off e_shoff;
  36 Elf32_Word e_flags;
  40 Elf32_Half e_ehsize;
  42 Elf32_Half e_phentsize;
  44 Elf32_Half e_phnum;
  46 Elf32_Half e_shentsize;
  48 Elf32_Half e_shnum;
  50 Elf32_Half e_shstrndx;
} Elf32_Ehdr;
```

In particular, according to the above ELF header definition, each ELF header has 52 bytes, and it specifies the file offset of the program header table (PHT) and section header table (SHT). As shown in Fig. 3, for our working example ls binary, the PHT (an array of program headers) starts at address 0x34 00 00 00 (52 bytes into file) and SHT (an array of section header) starts at address 78 64 01 00 (91,256 bytes into file). It also tells us that this binary code...
has 8 program headers with 32 bytes each, 28 section headers with 40 bytes each, and the section header string has 8 program headers with 32 bytes each.

Section headers do have a unique signature. Specifically, using the section header data structure defined below:

```
typedef struct {
    Elf32_Word sh_name;
    Elf32_Word sh_type;
    Elf32_Word sh_flags;
    Elf32_Word sh_addr;
    Elf32_Off sh_offset;
    Elf32_Word sh_size;
    Elf32_Word sh_link;
    Elf32_Word sh_info;
    Elf32_Word sh_entsize;
    Elf32_Addr sh_require_end;
} Elf32_Shdr;
```

From the documentation of ELF structure (Executable and linkable format), we check the following:

- \( V(sh_name) \in \{0, \ldots, 255\} \) for the first 4 bytes. This is because, since the section string table is usually less than 256 bytes, the string table index, \( V(sh_name) \) should be within \( \{0, \ldots, 255\} \).
- \( V(sh_type) \in \{0, \ldots, 11\} \) or it takes the other four special values such as 0x70000000 and 0x7fffffff.
- \( V(sh_flag) \leq 0x1.0x2.0x4.0x00000000 \).
- \( V(sh_addr) \) aligned with 4 bytes.
- \( V(sh_off \{f,s\}) \leq L(footer) \).
- \( V(sh_size) \leq L(footer) \).
- \( V(sh_link) \) and \( V(sh_info) \) could be 0, or \( I \) which is an index value to SHT, string table, or dynamic symbol table.
- \( V(sh_addralign) \) is 0 or positive integral power of two.
- \( V(sh_entsize) \) is 0 or \( E \) which is the size of the corresponding entry.

From our profile, \( I \) is usually less than 13, and \( E \) is less than 11. With these value constraints, and the total number of section headers (28 in our working example), we are able to accurately locate our footer.

**Searching PHT**

A PHT is used to locate the segments. In our example, the section header starts at 0x34. Usually, a program header starts right after the ELF headers (it will be in the same 4K block). We do not need to search to locate this. However, if necessary, we can derive the program header signatures according to its definition below, to scan for them in a manner similar to section header signature derivation.

```
typedef struct {
    0x0 Elf32_Word sh_type;
    0x0 Elf32_Word sh_flags;
    0x0 Elf32_Off sh_offset;
    0x0 Elf32_Word sh_addr;
    0x0 Elf32_Word sh_size;
    0x0 Elf32_Word sh_info;
    0x0 Elf32_Word sh_entsize;
    0x0 Elf32_Addr sh_require_end;
} Elf32_Shdr;
```

Fig. 3. Data layout of our Ls binary.

Fig. 4. Program headers of our Ls binary.
From the program header, we first infer the base virtual address of the image file while loading into the memory. For example, as shown in Fig. 4, the first segment PHDR starts at offset 0x34 with Addr 0x8048034. We can therefore conclude that the base virtual address is 0x8048034 – 0x34 = 0x8048000. More importantly, we can also learn each segment offset in the file, for example, INTERP segment starts at 0x134 and its content contains a string /lib/ld-linux.so.2. All other program code segments such as .init, .pl, and .text sections are at offset 0x0 with size 0x15ea4. After that, there is a data segment including sections such as .ctors, .got, .got.plt, .data, .bss at 0x16000, with size 0x390. We keep iterating each program header and eventually build a road map of the layout of the binary code. The output of our road map containing the place holders for every $b_i$ except $b_0$ (i.e., $h_i$) is well-defined and filled. Our goal is to gradually fill in the content for each other $b_i$, where $i \in \{1, \ldots, l(footer)/Block\ Size\}$.

### 3.2. Block-node linker

From the ELF header (the header) we can locate PHT, and using the signature we can locate the SHT (the footer). If there is no fragmentation, we can directly scan the disk and link $b_i$ to its following $b_{i+1}$ provided that we can exclude the indirect data block (using some heuristics). However, if a garbage gap exists, such an approach will fail. Thus, the most challenging part in Bin-Carver is deciding how to link each individual $b_i$ assuming the worst case scenario that a large volume of blocks are fragmented.

As a result, we have to explore the internal semantic relation between $b_i$ and $b_j$ to “logically” connect them. For input such as text files, we may utilize word splitting heuristics to merge two blocks. However, we are facing binary code. Therefore an intuitive (but naive) approach may try to disassemble each block $b_i$ and from the disassembled code to explore the semantics between two blocks such as the program control flow paths. However, it is very challenging to disassemble each $b_i$ individually for x86 binary code because a legal instruction could start at arbitrary place in $b_i$.

Fortunately, we have a new observation: we can explore the address pair of caller–callee to fill the block place holder of caller $b_{\text{caller}}$ and callee $b_{\text{callee}}$ and logically “link” $b_{\text{caller}}$ and $b_{\text{callee}}$ together. For example, as shown in Fig. 5, for the instruction e8 de 00 00 00 at file offset 0x0149d (in $b_1$), its target address is encoded in the operand de 00 00 00 which is 0x0149d (the file offset) + 5 (instruction length) + Oxde (operand) = 0x01580. Then we can expect the target address at 0x01580 to be a function prologue. Note that a function prologue usually has a signature such as push ebp, mov esp, ebp (for local function calls within the binary code), or a PLT table with a jmp, push, jmp instruction sequence (for library calls). Thus, we can search for the function prologue at an offset of 0x580 from block $b_i$ ($i \geq 1$), and find it is at 0x01580 which is in PLT and still in $b_i$. Then, we could “link” the two blocks together though they are both $b_1$ in this case.

As another example, for the last call instruction e8 f7 f5 fe ff at file offset 0x11e84 (virtual address 0x8059e84) in Fig. 5, from the operand f7 f5 fe ff, we can infer the callee block from 0x11e84 + 5 + 0xfffef5f7 = 0x1480. Meanwhile, we have learned 0x1480 is in our second block $b_1$, and we can directly compute the $b_{\text{callee}} = b_{11}$ because (0x1480 – 0xfffef5f7 – 5)/4096 = 11. Then we can link $b_1$ to $b_{11}$ or $b_{11}$ to $b_1$ because once either node is determined, we can determine the other one. That is, the “logic” connection is bi-directional.

We could also observe that for the library call case (the first example), since we have learned the PLT block number, we can use the PLT block number as an anchor with which to identify the absolute block number of the calling block. For a local call case, we can only determine the relative distance between caller and callee block. Also, we need to emphasize that we only need to search for instruction sequences starting with e8 (the CALL opcode) without really disassembling the binary code. There are many e8 sequences scattered across blocks (Note that an unpacking framework Eureka (Sharif et al., 2008) also leveraged this observation for their bi-gram analysis). For example, we found 958 direct call instructions scattered across 18 blocks among the 23 blocks in total for 1s. But we cannot use indirect calls (machine code ff ff or 9a) because we cannot resolve the target address statically. Also, unlike a callee’s prologue-signature, we cannot use the jmp target because we cannot differentiate the target address since the jmp target could be anywhere (and there is no signature of the target as well!). Additionally, in most cases we can directly use library calls to resolve all the block numbers because the number of library calls is significantly larger than local calls (for example, it is 956 vs. 22 in the 1s binary).

### 3.3. Conflict-node resolver

After being processed by our block-node linker, a particular place holder $i$ could have several candidate blocks. We will have to eliminate the redundant ones. Our solution is to use the already identified non-conflict nodes, which we use to explore the internal logic connections and resolve the conflict node. For example, from our $b_0$ (there is only one), the ELF header node, we can resolve each segment and section if there are any constraints (such as, for a string table, we would expect that the target node contains strings). If there are more than two library calls in a block, the relative distance between the PLT and the caller target
should be identical if the PLT is located in one block. Also, with more and more nodes identified, there will be more caller and callee relations, and we will keep resolving until reaching a fixed point (no node can be removed further).

Meanwhile, our block-node linker only focuses on linking the code blocks of the ELF binary. One may wonder how we handle the other data blocks such as data in .data or .debug sections. Actually, our conflict-node resolver can facilitate this. Specifically, our current Bin-Carver design is that we treat data sections as one block that is between the ELF header and the first block of the code section we identified. If the data block is fragmented, our resolver will explore other constraints defined in the PHT and SHT to resolve them. For instance, the .rodata section will be defined in the SHT, and .rodata is usually strings. Thus, we can first resolve the location of the .rodata section in the disk. This will also help identify other sections. Various .debug sections, if present, show the same predictable and exploitable patterns. In the worst case, if the data section does not contain any of these identifiable sections and is fragmented, then we cannot recover the data section, and we have to resort to dynamic execution to eliminate the bogus permutations. That is, we have to try the permutations of the garbage block when generating the binary files to test. If the recovered binary file does not crash during execution, then that is one sign of successful recovery. One of our future efforts will investigate other possible solutions.

3.4. Putting it all together

When given a disk image D, Bin-Carver will first remove those indirect data blocks because they are data gaps. The signature to remove them is to look for the data block in which data is well-aligned (4 bytes), for a certain chunk it contains a contiguous block number (non-decreasing), and that there is no duplicated element in the entire block (data block is not shared). It is not entirely necessary to use this signature to filter the indirect block, as our linker and resolver is able to get rid of it in most cases. However, to make our system more efficient (run faster), we use this signature approach to filter the indirect blocks.

After that, we search all the ELF headers using our ELF header scanner. For each ELF header hi, we use the algorithm presented in Algorithm 1 to recover the ELF file ei. More specifically, we first search the PLT table using the signature described in §3.2 and locate its logical block number (line 2). Next, we find the end block of our file by invoking function SearchSectionFooter (line 3). After that, we initialize the set for each file content block from 0 to n with an empty set (line 4 and line 5), except B0 with the ELF-header block (line 6). Then we scan each physical block starting from ELF-header block (b0) to the maximum block bMax (line 7). If there are any library calls in the physical block (line 8), we will compute its logical block number using the PLT block number (line 9), and union the target block with this physical block (line 10). Our algorithm from line 5 to line 8 works similarly to the insertion sort algorithm (Knuth, 1997) in the sense that we both first scan the blocks and then insert the block at the right place.

Not every Bi gets filled with a block bn because bn may not have library calls. Then we explore the logic gaps in two selected blocks to fill the unselected blocks. For example, if B1 and B2 both have blocks but not Bn then we will fill the unselected “gap” blocks to fill Bn (HandleUnselectedNode in line 11). After that, a particular set Bi may have multiple b, then we will explore all other constraints as described in §3.3, to eliminate the redundant nodes in Bi (ResolveConflictNode in line 12). Eventually, we concatenate the block content from Bi to output the final ELF file (line 13).

4. Evaluation

To demonstrate the effectiveness of this approach, we have implemented Bin-Carver using C# for our algorithm 1. A mixture of C# and Python code was used to help collect statistics and produce disk images. The entire system consists of approximately 1700 lines of code.

It is worth noting that initially we planned to compare Bin-Carver with some other state-of-the-art file carving tools such as Foremost (Foremost: a console program to recover files) and Scalpel (Richard and Roussev, 2005). However, it turns out that neither of them support carving for fragmented ELF binary files. This again demonstrates that Bin-Carver is truly a first-of-its-kind tool that offers a unique file carving capability.

4.1. Experiment setup

Sample data collection To evaluate Bin-Carver, we created 8 disk images with 2G-bytes each, and they can be classified into two sets: one (from Disk-1 to Disk-4) is the disk images without any overwritten files, the other set (from Disk-5 to Disk-8) is the disk images with multiple copy and delete rounds. In particular,

Algorithm 1 ELF-file recovery

**Require:** LibCall(b) returns a list of the library calls in a block b; Dist(c, plt) returns the absolute block number of the caller block for a library call c when given PLT block number plt; SearchPLT(i, j) returns the block number of the PLT table in block b; and b1; p(x) returns the physical block content of x; SearchSectionFooter(b) returns the block number of the physical disk address of the SHT searched from block b.

**Input:** the disk image D (which has excluded indirect data blocks), an ELF-header hi, and the total logic block number n which is identified from either the section header offset in ELF header or the maximum file offset from the PHT.

**Output:** ei, an executable file for header hi.

```
1: ELF-Carver(hi, n, D) {
2:   K ← SearchPLT(0, n)
3:   Max ← SearchSectionFooter(b0)
4:   for jε(0, Max) do:
5:     Bj ← {}
6:     B0 ← {b0}
7:   for jε(0, Max) do:
8:     for each cε LibCall(bj) do:
9:       m ← Dist(c, K)
10:      Bn ← Bn U {b}
11:     HandleUnselectedNode(0, Max)
12:    ResolveConflictNode(n, Bn)
13:   e ← Concatenate(B0, Bn)
14: }
```
• Disk 1 – This disk contained the contents of /bin. This is a small baseline sample, as it contained 117 binaries at only around 10 MB in size.
• Disk 2 – This disk contained /bin, /sbin, and /usr/bin. This resulting in more ELF files, for a total of 1090 binaries.
• Disk 3 – This disk contained about half the binaries from disk 2 (545 files), with another half of that unlinked before the snapshot was taken, leaving 274 binaries on disk and 271 deleted binaries remaining in the raw data.
• Disk 4 – This disk contained the entirety of disk 2 as well as some SO ELF files from /lib, resulting in a total of 1265 binaries.
• Disk 5 – This disk was created by first copying all of the files from disk 4 into the disk, deleting all of them, and then randomly copying half back.
• Disk 6 – This disk was created by first copying all of the files from disk 4 into the disk, deleting half of them randomly, and then copying half of the files from disk 4 back into the disk.
• Disk 7 – This disk repeated the same cycle as the previous one 2 times with smaller batches to create a messier disk image.
• Disk 8 – This disk did more unpredictable, smaller copy and delete cycles to create the most chaotic image.

Some metrics on disk image creation (such as the number of copy and remove operations) are provided along in Table 1 to help clarify the operations that went into constructing each data point. It is important to note, however, that disk state is just as much a product of the context and order of the operations as it is the quantity, so the descriptions of each disk’s creation are important to understand as well.

More specifically, we use the following nine metrics (showing from the 2nd column to the 10th column in Table 1) to describe the characteristics of these disk images. In particular,

1. \( \sum \text{Copies} \) – The total executions of the \texttt{cp} command used to build the disk.
2. \( \sum \text{Removes} \) – The total executions of the \texttt{rm} command used to build the disk.
3. \( \sum \text{Operations} \) – The sum of all operations applied to the disk.
4. \( \sum \text{ELF}\_F \) – The total number of ELF files on the disk. This is a count of the valid ELF files on the hard drive before it was unmounted and the image was created.
5. \( \sum \text{Del}\_F \) – The total number of ELF files deleted from the disk. This is the number of ELF files which were originally on the hard drive, but were deleted at some point before the image was created.
6. \( \sum \text{Frag}\_F \) – The total number of fragmented ELF files. This is the number of ELF files whose data block sequence contains one or more non-contiguous block pairs. In Fig. 6, this would be 3 for both collections because each one contains 3 fragmented files.
7. \( \sum \text{Breaks} \) – The total number of breaks in ELF data block sequences for the entire disk. This gives an idea of how many file fragments exists. For example, 2 files in 2 pieces each would result in a \( \sum \text{Breaks} \) metric of 2, but 2 files in 4 pieces each would result in a \( \sum \text{Breaks} \) metric of 6. In Fig. 6, file system F would have a score of 3, while file system P would have a score of 6.
8. \( G_g \) – The average garbage blocks per file among all fragmented files. This is \( \left(1/|F|\right)\sum_{i=1}^{j} |g_i| / |F| \), where \( |F| \) is the set of fragmented files and \( g_i \) is the set of garbage blocks in file \( i \). In Fig. 6, \( F \) would have a score of 20.3 for this metric, while \( P \) would have a score of 2.0.
9. \( G_G \) – The average garbage blocks per file among all files in the file system. This is \( \left(1/|D|\right)\sum_{i=1}^{j} |g_i| / |D| \), where \( |D| \) is the set of fragmented files and \( g_i \) is the set of garbage blocks in file \( i \). Since all files in Fig. 6’s file systems are fragmented, this value will be the same as \( G_g \).

Note that all files in our system are ELF binaries. Thus, our sample data is the worst case for our recovery because all the noisy data blocks are likely to contain machine code, which significant challenges our system, especially our block-node linker and conflict-node resolver components, due to high false positive rates. We ran some tests to verify this assumption and noticed no difference in recovery accuracy between comparable disks that differed only in the addition of heterogeneous data. We tested by filling a disk to capacity with PDF, MP3, MS Office, and image files. We then deleted enough to copy the 117 ELF files that will be used by Disk 1. No change in behavior was observed.

Each testing disk image size is 2G-bytes to help cut down on variation of carving results due to disk size. The algorithm can scale up to larger disks if a few locality

<table>
<thead>
<tr>
<th>Disk image</th>
<th>Disk creation</th>
<th>Disk characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk 1</td>
<td>117 0 117</td>
<td>117 0 38 38 1.0</td>
</tr>
<tr>
<td>Disk 2</td>
<td>1090 0 1090</td>
<td>1090 0 389 390 1.002571 0.357</td>
</tr>
<tr>
<td>Disk 3</td>
<td>546 273 819</td>
<td>545 271 190 191 1.005263 0.3498</td>
</tr>
<tr>
<td>Disk 4</td>
<td>1265 0 1265</td>
<td>1265 0 466 467 1.002164 0.3689</td>
</tr>
<tr>
<td>Average</td>
<td>754.5 68.25 822.75</td>
<td>754.25 67.75 273 271.5 1.0025 0.349423</td>
</tr>
<tr>
<td>Disk 5</td>
<td>1897 1265 3162</td>
<td>632 633 449 1449 1.004474 0.3546</td>
</tr>
<tr>
<td>Disk 6</td>
<td>1897 633 2530</td>
<td>934 317 502 1678 1.161148 0.415</td>
</tr>
<tr>
<td>Disk 7</td>
<td>1771 378 2149</td>
<td>681 584 453 800 1.5364 0.5487</td>
</tr>
<tr>
<td>Disk 8</td>
<td>1097 981 2078</td>
<td>772 319 785 955 2.065789 0.7195</td>
</tr>
<tr>
<td>Average</td>
<td>1665.5 814.25 2479.75</td>
<td>754.75 463.25 547.25 1220.5 1.44195 0.5097</td>
</tr>
</tbody>
</table>
assumptions are made. When the distance to look for section signatures is constrained to a constant, the performance of the algorithm for each individual file is invariant with respect to the size of the disk. This means that the algorithm’s performance relies only on the number of files to be recovered rather than the size of the disk.

**Ground truth collection** To evaluate how accurate our system is, we also need to collect the ground truth; we have to verify that our recovered files are the true ELF files. This turns out to be a challenging problem since we are performing reverse engineering. Fortunately, as when we create the disk image, we have the ground truth of the true ELF files in terms of the file contents. Thus, we can create an MD5 hash of both the first block as well as each individual block of the entire file for each true ELF binary (hashing each individual block of a file allows us to detect the true data in highly fragmented scenario), and then, for each recovered binary whose first block MD5 exists among the true ones, we can compare each individual block MD5 to ensure that the integrity was kept. This ground truth collection is in fact a file fingerprinting based recovering technique (McDaniel and Heydari, 2003).

### 4.2. Effectiveness

We tested Bin-Carver with the above 8 disk images for the effectiveness, in terms of the identification rate and recovery rate. More specifically, these two metrics are described in the following:

1. **Identification Rate (IR)** – IR shows the portion we can still identify in the disk no matter how fragmented the disk is. It is defined as the proportion of valid files in the file system that were identified before its image was taken. For cases such as Disk 3 in which deletions occurred but no further operations were carried out, the deleted files are considered valid. In other cases, deleted files are not considered valid if the hash value cannot match the original files, as further copy operations may have overwritten part of the latent file and this will be detected by our block-based hash approach. Issues such as malformed files, extreme fragmentation, or missing Shared Object metadata can all degrade the identification ratio.

2. **Recovery Rate (RR)** – This is the proportion of valid files in the file system before its image was taken that were identified and successfully recovered. This metric directly shows how effective our system is with respect to the identified files.

Our effectiveness evaluation results are presented in Fig. 7. We can see that our Bin-Carver has met our expectations: it performs extremely well in extracting ELF binaries without metadata with an IR of 96.3% and RR of 93.1% on average when handling file systems ranging from pristine to chaotic highly fragmented. Additionally, on disk images 1, 2, 3, and 4, it has predictably solid low failure rates (99.85% IR, 98.1% RR). The similarity between disk 2 and 3 demonstrates that, when they remain in the disk intact, the deleted ELF files are handled without a problem.

Disk 4 only suffers a slight dip compared to disk 2, and this is due to the unreliable nature of the Shared Object files, which are not even required to have section header tables (SHT). Disks 5 through 8 show far greater fragmentation, as they were a result of attempting to contrive pathological bad cases in which not only is there heavy fragmentation, but in which the fragmentation consists of other ELF files. The disks from 5 to 8 show predictably degraded effectiveness as the disks were subjected to an increasingly lengthy battery of copy and delete operations, and disks 7 and 8 showed the greatest decline in the effectiveness due to very messy file systems. However, Bin-Carver still manages to recover a majority of the ELF files intact from the extremely chaotic images.

The failed recovery in the non-fragmented images are due to occasional files that do not match expectations. We manually examined the causes and we found these were mainly due to SO libraries that do not have section header tables, or due to the malformed ELF files. Also, there were several ELF files that, upon manual inspection, did not seem to contain data that matched their header in structure. These files obviously invalidate many of the structural assumptions made during recovery and will, as a result, not be recovered.

### 4.3. Performance overhead

We also tested the run-time overhead of our Bin-Carver against these disk images. The data is presented in Fig. 8. We see that the performance is split into two groups. The first three finished almost immediately, while the
To identify PHT, currently we assume it is always right. We will investigate how to address this problem systematically.

The only problem is just that the ELF-footer is located in other places because ELF (Executable and linkable format) vs. IAT (import address table in PE). Our immediate concern performance considerations rather than resilience toward fragmentation. Additionally, they are geared toward recovering multimedia files and very little effort has been put forth toward handling binary executables, which actually directly motivated our Bin-Carver.

Carver (Garfinkel, 2007) attempted to solve this fragmentation problem by examining files as potentially split into a pair indicated by the file’s header and footer. Once these endpoints are identified, different combinations of fragments pairs are exhaustively checked until the file passes some decoder’s verification algorithm. While this header-footer and verifier approach has made a large step in file carving, Anandabrata et al. (Pal et al., 2008) illustrated several problems with it. For example, this approach sometimes suffers from poor scalability: a bi-fragmented file with a large gap in between will take too long to finish. Also, validation using decoding is frequently subject to false positives or impossible to apply to a particular file type.

Another fragmentation-resilient carving approach was proposed by Pal and Memon (Pal et al., 2003) that viewed the problem graphically. They used a modified Dijkstra’s algorithm (Dijkstra, 1959) to find a shortest path between a graphical representation of the fragments. It makes a greedy choice using a weight function defined using a metric of “edge similarity” to rate transitions by the compatibility of the concatenation of the contents at the end of the start node and the beginning of the end node. An improvement was later proposed (Pal et al., 2008) for this approach, which uses a sequential hypothesis test to move through the file block by block and attempt to pinpoint the beginning and end of fragments using statistical hypothesis testing.

Recent efforts in memory forensics also explored the graph-based approach to recover data instances. In particular, exploiting the points–to relation defined in data structure definitions, SigGraph (Lin et al., 2011) shows that...
we can derive a robust graph-based signatures in identifying kernel data instances. DIMSUM (Lin et al., 2012) further pushes this graph-based approach in identifying deleted data objects.

Bin-Carver shares the idea of exploring the graph connection between data blocks, but differs in the way of how to find the connections. In particular, previous work did not attempt to identify the signatures in the binary code such as our verifiable caller–callee signatures to connect the data blocks. Moreover, they did not make use of any “road map” within files and instead rely on edge compatibility.

Finally, besides the above file carving technique, as demonstrated in our experiment, we can use hash value (MD5) based technique to identify the data block in disk for known files, which was proposed by McDaniel and Heydari (McDaniel and Heydari, 2003). There are also other approaches, such as byte frequency based classification and clustering technique (Karresand and Shahmehri, 2006a, 2006b; Moody and Erbacher, 2008). Bin-Carver complements these techniques by exploring other features inside file content for the file recovery.

7. Conclusion
We have presented Bin-Carver, a tool for dissecting, mapping, and recovering binary executable files from raw binary data. The key idea is to explore the road map information defined in executable file headers and the explicit control flow paths present in the binary code, to “logically” connect the data blocks in the disk. Our experiment with thousands of binary code files has shown that our Bin-Carver is extremely accurate, and much better than all the existing file carving techniques when recovering binary files with fragmentations. In addition, Bin-Carver also provides a useful complement to the more traditional header-footer pairing approach for file carving to gain more complete disk image recovery.

Acknowledgment
We greatly appreciate our shepherd Golden G. Richard III for his tremendous help in improving the paper. We also thank the anonymous reviewers for their insightful comments and suggestions. This research was supported in part by the Air Force Office of Scientific Research (AFOSR) under Award No. FA-9550-08-1-0260. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of the AFOSR.

References
Executable and linkable format, http://wiki.osdev.org/ELF.