Formalizing the Honeynet

Defining and Proving a Rootkit Installation

CM Lubinski
DePaul University
cm.lubinski@gmail.com

Abstract

Definitions for multiple file formats, "malicious" file names, and forensic time lines are provided in Coq. These are combined to mimic the types of evidence given by independent forensics researchers in a Honeynet forensics competition. The Honeynet examples are then provided in the form of formal proofs based on these definitions. Along the way, functions for creating relevant data structures within Coq are also described.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

After completing a lengthy customs process in the largest airport of burgeoning global power, you notice that your laptop appears to be running rather slowly. You begin to suspect that a root-kit has been installed, but you are uncertain how to prove this. At first, you consider using off-the-shelf antivirus software to find the infection. The risk of damaging international relations between your countries is too great when using such proprietary software, however. What you really need is a concrete definition of a root kit and evidence fitting that definition to prove that your machine has been attacked.

This scenario is the ultimate goal of the research provided in this paper. We need formal (we will use the Coq programming language) definitions for various types of evidence needed by forensics analysts. We will also need a way to convert real data (e.g. from a disk image) into these definitions. Combining these two ideas, we can provide concrete *proofs* regarding whether or not a particular image fits such a definition; in the above example, we could provide a *proof* that a root-kit was installed.

To guide our search, we will focus on the evidence structures and proofs described by several independent researchers studying a "Honeynet" challenge. The Honeynet Project[1]'s now-defunct *Scan of the Month* series provided researchers a disk image attained from a compromised honeypot (a computer created with the explicit goal of catching malware for inspection.) Each month, they were challenged to describe what happened to the system and provide evidence for their conclusions. We will consider one specific example[2], in which a rootkit was installed on a server and the security community was asked to recover the rootkit, prove that it had been installed, and provide a step-by-step writeup describing how the rootkit was found.

2. Getting Our Feet Wet with File Types

We start by considering a relatively straight-forward request: defining what it means for a given file to be a JPEG. How can we formalize this notion? One tact (used by many operating systems) is to rely on the file extension (if present) – in this case, checking for either `jpg` or `jpeg`. This is a very loose definition, however, as malicious users would need give their files a different extension to avoid detection. We could instead review the JPEG spec and confirm that all of the meta data contained within this file is consistent with said spec. This approach runs the opposite end of the spectrum, requiring significantly more evidence. Further, the JPEG spec is not as tidy as we might hope, and most applications are lenient with the file formats they accept.

Instead, we will chose a middle route, opting to use "magic numbers" as our guide. This refers to tell-tale byte values at predictable points within the file data. These file signatures are *relatively* unique to various file formats, so we will use them in our definitions and add additional checks as necessary. JPEGs happen to always start with the bytes `ff d8` and end with `ff d9`. Similarly, gzipped files begin with `1f 8b 08` and Linux executables (ELFs) begin with `7f 45 4c 46` (`7f` E L F).

If we represent each byte as a positive integer, then writing these definitions in the proof-centric language of Coq (plus some syntactic sugar) would look like

```
Definition isJpeg (file: File) :=
     file @[  0 ] = value 255
  ∧  file @[  1 ] = value 216
  ∧  file @[ -2 ] = value 255
  ∧  file @[ -1 ] = value 217.

Definition isGzip (file: File) :=
     file @[ 0 ] = value 31
  ∧  file @[ 1 ] = value 139
  ∧  file @[ 2 ] = value 8.

Definition isElf (file: File) :=
     file @[ 0 ] = value 127
  ∧  file @[ 1 ] = value 69
  ∧  file @[ 2 ] = value 76
  ∧  file @[ 3 ] = value 70.
```

For each definition, the first (and, in the case of the JPEG, last) bytes of a file must both be present (as indicated by `value`) and equal to the byte sequences described above. We will discuss the absence of byte values later, but for now, assume that this accounts for "missing" evidence. By defining file types in this manner, we can use these definition as building block within larger definitions and within proofs. For example, given these definitions, we can *prove* that JPEG files cannot also be gzipped files:

```
Lemma jpeg_is_not_gzip : forall (file: File),
  (isJpeg file) → ¬(isGzip file).
```

A full version of this proof (and others mentioned throughout this paper) is provided in the appendix.

3. Expanding Definitions with Honeynet Examples

Let's now consider the Honeynet *Scan of the Month* mentioned in the introduction. In his entry for this contest, Matt Borland[3] described a deleted, "tar/gzipped file containing the tools necessary for creating a home for the attacker on the compromised system". Formalizing this a bit, we will say that he proved that

```
Lemma borland_honeynet_file:
  exists (file: File),
  (isOnDisk file honeynet_image_a)
  ∧ isDeleted file
  ∧ isGzip file
  ∧ exists (filename: ByteString),
     (In filename (Tar.parseFileNames (gunzip_a file)))
     ∧ looksLikeRootkit filename.
```

That's a bit of a mouthful, but we are stating there exists a deleted, gzipped file on the Honeynet disk image such that, when we unzip the file, the contained tar includes a malicious-looking file name. Note that both the Honeynet image and the unzipping operation are appended with '`_a`', which we will use to indicate "assumption". We will discuss assumptions in detail momentarily; we can treat them as another type of definition until then. We would instead prefer to focus on each of the definitions in the and clause (save `isGzip`, which has already been described).

3.1 isOnDisk

How might we define that a file "exists" on a particular disk image? For a simple definition, we could start by claiming a "file" is on a disk image if that file's contents could be found sequentially in the disk. In other words, we might consider the file to be on disk if we can find a starting index on the disk such that every byte afterwards matches that of the file.

```
Definition isOnDiskTry1 (file: File) (disk: Disk):
  exists (start: Z), forall (i: Z),
  (i >= 0 ∧ i < file.(fileSize)) →
    file @[ i ] = disk (start + i).
```

This definition isn't very useful in practice, however, as files are very often fragmented across multiple segments of a disk image. Moreover, this definition would also include false positives, where a "file" is formed by looking at the bits that span a fragment boundary. Files are stored on disks via file systems, which make no sequential guarantees (particularly in recent file systems such as ZFS and Btrfs).

We must therefore, build our definition for file existence with file systems in mind. Proving that a file exists within each type of file system is a relatively unique operation, however. As a result, we will define file existence as the disjunction of file existence within each file system. While the definitions are distinct, they each tend to follow the pattern that there exists some file identifier such that if we were to parse the file associated with that identifier on the given disk, we would find a file identical to the provided file.

```
Definition isOnDisk (file: File) (disk: Disk):
  (* Ext2 *)
  (exists (inodeIndex: Z):
    (Ext2.findAndParseFile disk inodeIndex) = value file)
  ∨ (* FAT32 *)
  (exists (clusterNumber: Z):
    (Fat32.findAndParseFile disk clusterNumber)
      = value file)
  ∨ (* Btrfs *)
  (exists (key: Z):
    (Btrfs.findAndParseFile disk key) = value file)
  ∨ ...
```

The functions `findAndParseFile` perform the work of inspecting the disk image, reading the relevant data structures, and creating an abstract representation of the results (in this case, a File object.) We will discuss parsing computations a bit later.

3.2 isDeleted

Like existence on the disk, what it means for a file to be deleted is quite file system specific. In Ext2, a file might be seen as deleted if its bit in the allocation bitmap were zero; in FAT32, we might consider a file deleted if it were not accessible via a cluster chain. To account for these varieties, we include will include the deletion status of a File within its definition. This definition attempts to describe "universal" file truths:

```
Structure File := mkFile {
  fileSize: Z;
  deleted: bool;
  byteOffset: ByteData
}.
```

The first two fields are self-explanatory, but the third warrants additional explanation. This field is a function that, when given an offset within a file, returns the byte value found at that offset within the file. As described in our investigation of file types, we must account for the possibility that the byte requested is not available (e.g. out of range or not within our assumptions). We must wrap the result in an Exc (a.k.a. Option) to handle this situation.

Coming back to goal, it should be easy to see that the definition of `isDeleted` effectively just delegates to the boolean value that is part of the File representation:

```
Definition isDeleted (file: File) :=
  file.(deleted) = true.
```

3.3 Malicious Filenames

The final clause in our lemma states that one of the filenames contained in the gzipped-tar is suspicious. We start with another assumption, in this case the decompression algorithm for gzipped files; while we try to keep as many of our computations within Coq as possible, some (like unzipping a file) were far out of scope, so we keep them as assumptions. Once the file is unzipped, we run another parsing computation, `Tar.parseFileNames`, this time retrieving all of the file names contained within the unzipped tar.

The only new *definition* is that of `looksLikeRootkit`. For the sake of this definition, we will define that a filename looks like a filename involved in a rootkit if that filename, excluding its path prefix, matches any of a predefined list of system files. These include task managers such as `top` and `ProcMon.exe`, which root kits replace to hide their activities, as well as `ssh` and `rsync`, which root kits replace to allow them the ability to monitor traffic. We treat file names as lists of bytes to account for unicode and other non-ascii characters which would be excluded by Coq's string/ascii package.

```
Definition looksLikeRootkit (fileName: ByteString):
  In (trimFileNamePrefix fileName)
     (map ascii2Bytes ("ps" :: "netstat" :: "top" :: "ifconfig"
                       :: "ssh" :: "rsync" :: "ProcMon.exe"
                       :: nil)).
```

Note that, for ease of description, we provide file names as ascii strings and then convert each to their equivalent, byte representation via the `ascii2Bytes` function.

Use of this definition is straight forward:

```
Lemma rootkit_ex1 : looksLikeRootkit (ascii2Bytes "last/top").

Lemma rootkit_ex2 :
  ¬(looksLikeRootkit (ascii2Bytes "last/example.txt")).
```

4. Lemmas, Computations, and Definitions; Oh My!

We have now come across several core concepts of this research, and it is easy to see how they might be confused. Before we continue, let us solidify our understanding of each category.

We are most familiar with **definitions**, which provide a name for a common understanding of a concept. For example, we provided definitions for `isGzip`, `looksLikeRootkit`, etc. The wider goal of this research is to provide a set of common definitions for concepts pertinent to the forensics community. This paper describes several which are relevant to the Honeynet example, but other definitions might include "web page access" (e.g. by inspecting browser history), "in contact with" (e.g. there exists records of email communication), and "last time logged in".

Definitions are most often built by combining aspects of various abstract **data structures**, "universal" representations of data-related concepts within forensics. We have seen these used to represent files, and we will see them represent various Ext2 file system structures, as well as events in a reconstructed timeline. These structures serve as a way of differentiating particular parsing computations (which build the structures) from definitions involving these structures. It should be relatively simple for someone to write a different parsing algorithm which produces the same structures without affecting any of the definitions.

Unfortunately, the Coq proof environment is quite limited with regards to practical applications. It cannot, for example, read bytes from a disk image; it cannot (for our intents) call external programs to transform data; it cannot even instantiate a list of tens of thousands of values. We work around these limitations by making **assumptions** about our environment (with the suffix `_a`) which entail only the relevant pieces of data needed for our proofs. We won't read a file from disk; instead, we will generate a *sparse* map to represent that disk, including only the offsets we care about. Assumptions are also used to account for transformations that would normally be performed by a trusted, external program. We cannot, for instance, run "gunzip", nor do we want to try to implement that algorithm within Coq. Instead, we require an assumption be made that encompasses that activity. Our proofs cannot validate that `disk_a` or `gunzip_a` accurately represent values on the disk or the output of an unzipping operation – they instead trust those assumptions to be accurate and rely on users for verification.

Where possible, we do attempt to **parse** and/or **compute** values within Coq. Here we have codified algorithms which take the bytes from disk (assumptions) and convert them into relevant structures. This might mean creating Inodes, SuperBlocks, and GroupDescriptors for a disk with Ext2 on it, or reading headers from the data associated with an abstract tar file. In all cases, these operations are our (perhaps flawed) implementations of various specifications; users should feel free to check the structures created against those found using other tools. Ultimately, we aimed to provide definitions and data structures which could serve as building blocks for proofs; parsing and computations are simply the glue that tie those building blocks to actual data.

The Honeynet competition we've described required researchers to provide "**evidence**" or "proof" that the disk had been compromised by a rootkit. This required that the researchers both define what acceptable evidence would look like as well as provide it the evidence regarding the attacked server. This led each researcher to provide different types of evidence depending on what that researcher found to be acceptable. While we could create definitions for the various types of evidence, we will instead follow the Honeynet applicants' lead and provide only the evidence itself in the form of a **lemma** or proof. This evidence will be in the form of a proof term, which the author of the proof can build up using Coq's tactic system. It is then up to the reviewer to determine whether or not the evidence provided by an applicant fits a familiar definition.

Finally, the code written for this research is sprinkled with **utility functions** to perform (mostly) simple transformations. `trimFileNamePrefix` and `ascii2Bytes` are two such examples which make explaining (and hopefully, understanding) the code samples easier. These functions are not necessary for the definitions, proofs, etc. where they appear; they simply make the code more terse.

5. Computing Our First Lemma

Not that we have introduced each of these concepts, let's step through our `borland_honeynet_file` proof, which will touch on each.

5.1 A Note on Functional Idioms

Particularly while parsing, we will make use of several idioms from functional programming. In particular, we make heavy use of the "option" pattern; as the parsing code does not know whether bytes will be *present* on the provided disk image, each attempt to read returns an *option* of the result. An option is simply a wrapper around a value such that the wrapper may be contain the value (indicated by `value` or `Some`), or may not contain the value (indicated by `error` or `None`). This is why the return type of many functions is `Exc A`; this indicates an option of type `A`.

Options make error handling "percolate up" in that functions which produce options can be chained such that any error results in the entire computation being an error. To get to this point be define two functions, indicated by the infix notation `_map_` and `_flatmap_`. The function signatures for each should aid their explanation.

```
Definition opt_map {A B: Type} (opt: Exc A) (fn: A → B)
  : Exc B.

Definition flatmap {A B: Type} (opt: Exc A) (fn: A → Exc B)
  : Exc B.
```

The role of `_map_`, then, is to transform the contents of an option. If the option is empty, map has no effect. `_flatmap_` similarly does not affect empty options, but performs a bit differently with `values`. Instead of simply transforming the contents, the entire option gets replaced with the result of `fn`, allowing sequences like

```
(file @[ 0 ]) _flatmap_ (fun byte0 ⇒
(file @[ 1 ]) _flatmap_ (fun byte1 ⇒
(file @[ 2 ]) _map_ (fun byte2 ⇒
  (byte0, byte1, byte2)
)))
```

which can return either `error` or `value` with all three bytes present. If the first byte was not present, the outer-most `_flatmap_` would not have executed the inner function. Similarly, if the second byte were not present, the function including `_map_` would not be ran, and if the third byte were unavailable, the function with the parameter `byte2` would not be evaluated.

We've rushed through these concepts as we assume the reader has some background in functional programming. For a more thorough (yet brief and practical) introduction, see Wamper & Miller[6].

5.2 Assumptions

Our proof will make use of two assumptions: `honeynet_image_a`, a `Disk` and `gunzip_a`, a function, `File->File`. The `Disk` type is used to represent a disk image throughout our definitions and proofs. Technically, they are functions from `Z` to `Exc Z`, acting as byte-retrieval mechanism; give a `Disk` an offset, and it will respond with the byte value (a `Z`) at that offset in the disk image, or `error` if no such byte exists on the disk. Generally, disk data is backed by an in-memory mapping via Coq's `FMapAVL` trees, and must be generated from the true disk values. Due to Coq's limitations on data structure sizes, we only populate these maps with essential values; once the map reaches a few thousand entries, it will no longer be usable within Coq.

The second assumption represents gzip's unzipping/deflating operation. As implementing the decompression algorithm used within Coq would not provide particularly valuable information, we instead delegate its operation to an assumption. This assumption needs to be able to convert a compressed, gzip file into the corresponding, uncompressed file, which effectively means providing a new `File` populated with the relevant bytes. In our example, we used the *Sleuth Kit*'s `icat` program to pull inode `23`'s contents from the disk image, ran `gunzip` on the resulting `tgz` file, and then pulled the relevant bytes (i.e. those necessary from the tar file headers). We effectively just wrap this data in a function that, regardless of `File` input, returns the uncompressed, tar `File`.

5.3 Parsing a File via INode

Our evidence relies on the *existence* of a `File` which is on disk, deleted, etc., so our proof for this evidence need only provide such a `File`. Hypothetically, this should be easy, as we can just pull the file from the disk image directly. We therefore use a parsing function (rather, a series of functions) to retrieve this file and provide it as demonstrative proof. Ultimately, we will need to call

```
(Ext2.findAndParseFile honeynet_image_a 23)
```

To be confident in this function's results, however, we will need to understand how Ext2 file systems are laid out on disk. We will proceed by pealing off each layer of the call and review the constructed data structures.

5.3.1 findAndParseFile

At the outermost conceptual layer, we have a function which, when given a disk and Ext2 Inode identifier, returns either an error or a valid File structure associated with that file. As we discussed earlier, a `File` is composed of a deletion status, a file size, and a function that retrieves bytes within the file. These fields require we first parse out the `SuperBlock` of the disk and the `GroupDescriptor` and `Inode` associated with our index (`23`).

To understand what these structures mean, one must first learn how Ext2 is structured. At its core, the file system is composed of a sequence of equally-sized chunks of bytes (called "blocks"). Files are not necessarily stored on sequential (or even contiguous) blocks; their data may be parcelled throughout the disk image (as we will describe when discussing Inodes, below). Each file is referenced by a single "Inode" structure, which keeps track of the file's data locations as well as access times, file size, and other meta data. Inodes are collected into "groups", which have meta data stored in a "Group Descriptor".

This descriptor includes information about which Inodes are allocated, etc. as well as provides a mechanism to segment the administration of Inodes. We can easily compute which group a particular Inode index belongs to by dividing it by the number of inodes per group. We can compute the Inode's position within that group by taking the remainder of this division (i.e. by applying the mod). It's important to note for both of these operations that Inode indices are one-indexed, as are group descriptors; this will lead to some additions and subtractions of one throughout our code to convert between zero- (which is easier to compute on) and one-based indices.

Meta data about the file system as a whole is stored in a special block known as the "SuperBlock", which lives at a predictable position on the disk. The SuperBlock contains information such as how large each block is, where the collection of GroupDescriptors starts, and the number of Inodes per group. While we do not use this fact, the SuperBlock is usually stored redundantly on the disk, meaning we could verify its values with the copies.

Returning to our parsing efforts we must note that each attempt to pull out a SuperBlock, GroupDescriptor, etc. may fail, and if this occurs, we want the entire File parsing function to propagate the failure. Here we will use the `_flatmap_` approach described above, treating the computation as a monad.

```
Definition findAndParseFile (disk: Disk) (inodeIndex: Z)
  : Exc File :=
  (findAndParseSuperBlock disk) _flatmap_ (fun superblock ⇒
  let groupId := ((inodeIndex - 1) (* One-indexed *)
                  / superblock.(inodesPerGroup)) + 1 in
  let inodeIndexInGroup :=
    (inodeIndex - 1) mod superblock.(inodesPerGroup) in
  (findAndParseGroupDescriptor disk superblock groupId)
    _flatmap_ (fun groupdesc ⇒
  (findAndParseInode disk superblock groupdesc inodeIndex)
    _flatmap_ (fun inode ⇒
  (parseDeleted disk superblock groupdesc inodeIndex)
    _map_ (fun deleted ⇒
    mkFile
      inode.(size)
      deleted
      (fetchInodeByte disk superblock inode)
  )))).
```

5.3.2 findAndParseSuperBlock

Regardless of block size, the first SuperBlock can be found starting at the 1024th byte. Below this position is the boot sector, executable code that loads prior to the main operating system (think boot loaders like LILO, GRUB, and MBR). While the size of the SuperBlock depends on the revision of Ext, the spec is largely compatible, so we will use the SuperBlock structure described by one of Ext2's original authors, David Poirier[5]. This spec cares only about the first 264 or so bytes of the block (regardless of its size).

Finding and parsing the SuperBlock, then, amounts to jumping to offset 1024 on the disk and plugging in the read values into a SuperBlock structure. To make our lives a tad easier, we will use a `shift` function, which effectively shifts the beginning of a disk by serving as a layer of indirection when pulling bytes from the disk. This wrapper allows us to parse as if the 1024th index were at position zero.

The values we need to store to construct a SuperBlock are largely encoded as little endian, 4-byte integers. To parse this sequence of bytes into a more usable integer, we will make frequent use of an unsigned conversion function, `seq_lendu`. This function simply reads in a sequence of bytes from the disk starting at a position (as given by the second parameter) and running for a specific length (as given by the third), and converts that into an integer based on little endian semantics. Note that in this code sample, we omit most of the SuperBlock's fields (there are roughly 45) as their addition should be clear.

```
Definition findAndParseSuperBlock (disk: Disk)
  : Exc SuperBlock :=
  let disk := (shift disk 1024) in
  (seq_lendu disk 0 4) _flatmap_ (fun inodesCount ⇒
  (seq_lendu disk 4 4) _flatmap_ (fun blocksCount ⇒
  (seq_lendu disk 8 4) _flatmap_ (fun rBlocksCount ⇒
  (* ... Additional fields omitted ... *)
  (seq_lendu disk 260 4) _map_ (fun firstMetaBg ⇒
    mkSuperBlock
      inodesCount
      blocksCount
      rBlocksCount
      (* ... Additional fields omitted ... *)
      firstMetaBg
  ))))))))))))))))))))))))))))))))))))))))))))))))).
```

While we parse out virtually all of the fields of a SuperBlock, we will need only a handful. `inodesCount` provides an upper bound for inode indices (which we use to validate that a given inode exists). `inodesPerGroup` will appear several times in this paper; the field indicates the number of inodes assigned to each GroupDescriptor. Hence, that field provides a way to determine which GroupDescriptor is needed for a particular inode index.

The `logBlockSize` field is also of interest, as we will use it to determine the number of bytes each block on disk spans. This is encoded using the logarithmic scale, but we provide a simple function to convert to the true number of bytes.

```
Definition blockSize (superblock: SuperBlock) :=
  Z.shiftl 1024 superblock.(logBlockSize).
```

5.3.3 Block Addresses

As mentioned earlier, the Ext2 file system breaks the disk into "blocks" for reference purposes; these are akin to cylinders in the FAT32 file system. Due to locality of reference, in practice, the need to retrieve a single byte from a disk is very rare. Instead, data is most often retrieved in sequential chunks, which maps well to the concept of data blocks. Depending on cache size, whole blocks are read at a time. These blocks are identified by their one-indexed "block address", signified by the type, BA.

To find the initial byte of a block based on its address alone, we need to first find the size of each block (defined in the SuperBlock.) With that, we can treat the disk as an array of block and simply jump to the relevant position.

```
Definition ba2Offset (superblock: SuperBlock) (blockAddress: BA)
  := (blockSize superblock) * blockAddress.
```

5.3.4 findAndParseGroupDescriptor

An array of GroupDescriptors can be found in the block following that which contains the SuperBlock. Depending on block size, that may mean the GroupDescriptors begin at block one if the block size is greater than 1024 (and hence, the SuperBlock is part of block zero) or at block two otherwise (when the SuperBlock composes all of block one).

GroupDescriptors will be represented by a structure similar to SuperBlocks, though with far fewer fields. GroupDescriptors have 32 allocated bytes, though only 20 are used. With this fact and knowledge of where the group descriptor array starts, we can jump to a particular GroupDescriptor by multiplying the structure's size by the index we seek. This formula needs to be modified slightly to account for one-based indexing, but otherwise the parsing is quite straight forward. As with the SuperBlock, we will shift the disk to aid our parsing efforts.

```
Definition findAndParseGroupDescriptor
  (disk: Disk) (superblock: SuperBlock) (groupId: Z)
  : Exc GroupDescriptor :=
  let groupBlockArrayBA := if (blockSize superblock =? 1024)
    then 2 else 1 in
  let groupBlockArrayOffset :=
    ba2Offset superblock groupBlockArrayBA in
  (* groupId is one-indexed *)
  let descriptorOffset := 32 * (groupId - 1)  in
  let disk := (shift disk (groupBlockArrayOffset
                           + descriptorOffset)) in
  (seq_lendu disk 0 4) _flatmap_ (fun blockBitmap ⇒
  (seq_lendu disk 4 4) _flatmap_ (fun inodeBitmap ⇒
  (seq_lendu disk 8 4) _flatmap_ (fun inodeTable ⇒
  (seq_lendu disk 12 2) _flatmap_ (fun gdFreeBlocksCount ⇒
  (seq_lendu disk 14 2) _flatmap_ (fun gdFreeInodesCount ⇒
  (seq_lendu disk 16 2) _flatmap_ (fun usedDirsCount ⇒
    mkGroupDescriptor
      blockBitmap
      inodeBitmap
      inodeTable
      gdFreeBlocksCount
      gfFreeInodesCount
      usedDirsCount
  ))))))).
```

Two of the fields from this structure will be particularly useful, `inodeBitmap` and the `inodeTable`. Both contain block addresses, pointing to the start of a sequence of inode-related data. The former is simply an array where each bit represents whether or not the corresponding Inode is allocated; this will be of great use when we are calculating deletion status. The latter points to an array of Inode structures, as described in the next section.

5.3.5 findAndParseInode

Ext2 tracks meta information about specific files through "Inode" structures. One such data structure exists for each file and contains copious information relevant to our interests, including creation time, deletion time, and references to the data blocks which make up this file. As each file has a unique Inode within the Inode array, we can refer to files by their *Inode Index*, which we do throughout this paper.

GroupDescriptors contain a reference to the block address associated with the start of the Inode array. From there, we can pin point the beginning of the relevant Inode structure by calculating the Inode's position within the block group the same way we calculated GroupDescriptors – each Inode is 128 bytes. Add in a one-based offset and a check that the requested Inode is valid and you have the `findAndParseInode` function.

```
Definition findAndParseInode (disk: Disk)
  (superblock: SuperBlock) (groupdesc: GroupDescriptor)
  (inodeIndex: Z) : Exc Inode :=
  (* Check for valid Inode *)
  if (inodeIndex >=? superblock.(inodesCount))
  then error
  else
    (* Inode Table is 1-indexed *)
    let inodeIndexInTable :=
      ((inodeIndex - 1) mod superblock.(inodesPerGroup)) in
    let inodePos := (ba2Offset superblock
                               groupdesc.(inodeTable))
                    + (inodeIndexInTable * 128) in
    let disk := (shift disk inodePos) in
    (seq_lendu disk 0 2) _flatmap_ (fun mode ⇒
    (seq_lendu disk 2 2) _flatmap_ (fun uid ⇒
    (* ... Additional fields omitted ... *)
    (seq_lendu disk 40 4) _flatmap_ (fun directBlock1 ⇒
    (seq_lendu disk 44 4) _flatmap_ (fun directBlock2 ⇒
    (* ... Additional direct blocks omitted *)
    (seq_lendu disk 84 4) _flatmap_ (fun directBlock12 ⇒
    (seq_lendu disk 88 4) _flatmap_ (fun indirectBlock ⇒
    (seq_lendu disk 92 4) _flatmap_ (fun doubleIndirectBlock ⇒
    (seq_lendu disk 96 4) _flatmap_ (fun tripleIndirectBlock ⇒
    (* ... Additional fields omitted ... *)
    (seq_lendu disk 116 4) _map_ (fun osd2 ⇒
      mkInode
        mode
        uid
        (* ... Additional fields omitted ... *)
        (directBlock1 :: directBlock2 :: directBlock3
          :: directBlock4 :: directBlock5 :: directBlock6
          :: directBlock7 :: directBlock8 :: directBlock9
          :: directBlock10 :: directBlock11 :: directBlock12
          :: indirectBlock :: doubleIndirectBlock
          :: tripleIndirectBlock :: nil)
        (* ... Additional fields omitted ... *)
        osd2
  ))))))))))))))))))))))))))))))))).
```

5.3.6 parseDeleted

With the Inode, we know the first attribute of our File object, the file size. We next turn to the second field, whether or not the file is marked as deleted, or "unallocated". To find the allocation status of an Inode, we'll need to find the associated allocation bitmap (a sequence of bits on disk such that zero indicated unallocated and one indicates allocated.) The bitmap associated with a particular Inode is indicated in that Inode's GroupDescriptor. As we are reading entire bytes at once, we'll read the byte which contains the allocation bit for the Inode we care about and then test the bit within the read byte.

Ultimately, this means we will find the start of the allocation bitmap from the GroupDescriptor, jump to the byte within that bitmap which contains the allocation of our Inode, and then test the bit within that byte. As noted before, we need to be weary of the fact that Inodes are one-indexed.

```
Definition parseDeleted (disk: Disk) (superblock: SuperBlock)
  (groupDesc: GroupDescriptor) (inodeIndex: Z) : Exc bool :=
  let inodeIndexInGroup :=
    (* 1-Indexed *)
    (inodeIndex - 1) mod superblock.(inodesPerGroup) in
  let bitmapStart :=
    ba2Offset superblock groupDesc.(inodeBitmap) in
  (* Fetch the allocation byte for this inode *)
  (disk (bitmapStart + (inodeIndexInGroup / 8)))
    _map_ (fun allocationByte ⇒
    (* The bit associated with this inode is 0 *)
    (negb (Z.testbit allocationByte
                     (inodeIndexInGroup mod 8)))
  ).
```

5.3.7 fetchInodeByte

With the deletion status and file size, we need only more more attribute to create our File structure. File objects have a method which, given an arbitrary offset within the File, returns the associated byte within the File, effectively linearizing byte access. This abstraction is quite necessary as each File system will have a different method of accessing the File bytes. We now discuss how such a method works on Ext2.

Ext2 was designed with a trade-off in mind; quick access to file data generally runs counter to the desire to store very large files. The solution the file system employs is to allow the first twelve data blocks to be directly addressable from the Inode. Slightly larger files can be addressed through an indirection pointer, which points to a block which in turn points to several blocks of data. Even larger files can be addressed through a double-indirection pointer, which points to a block which points to other blocks which each contain references to additional data blocks. There is also a third level of indirection, allowing Ext2 to address up to roughly sixteen million data blocks.

This means that determining which block a given byte within a file is located in may require walking one or more indirection blocks. As this is a recursive operation (depending on the level of indirection,) we next describe a fixpoint which will pull out the block address associated with a given byte. This function checks if it has reached the final level of indirection, as indicated by the `O` (Coq's zero) case; if so, it knows to grab the correct block address (a four byte value) from the block of pointers.

If the function has not reached its base case (`S`, the "successor of"), it determines which of its block addresses to follow, reduces the block number requested accordingly, and recursively tries again. To select the correct block address to recursively follow, we must determine how many blocks each indirection pointer is responsible for (`unitSizeInBlocks`), and select the index of the block address within the array accordingly (`nextBlockIndex`). Using that, and the fact that each address is four bytes large, we jump to the correct offset within the pointer array, read those four bytes, and recurse accordingly.

```
Fixpoint walkIndirection (disk: Disk) (superblock: SuperBlock)
  (blockNumber indirectionPos: Z) (indirectionLevel: nat)
  : Exc Z :=
  match indirectionLevel with
  | O ⇒
    let bytePosition := (indirectionPos + 4 * blockNumber) in
    (seq_lendu disk bytePosition 4)
  | S nextIndirectionLevel ⇒
    let exponent := Z.of_nat indirectionLevel in
    let unitSizeInBlocks :=
      ((blockSize superblock) ^ exponent) / (4 ^ exponent) in
    let nextBlockIndex := blockNumber / unitSizeInBlocks in
    let nextBytePosition :=
      indirectionPos + 4 * nextBlockIndex in
    (seq_lendu disk nextBytePosition 4)
      _flatmap_ (fun nextBlockBA ⇒
      walkIndirection disk superblock
                      (blockNumber mod unitSizeInBlocks)
                      (ba2Offset superblock nextBlockBA)
                      nextIndirectionLevel
    )
  end.
```

Building on top of this fixpoint, we can define a single function that, when given a Disk, SuperBlock, Inode and offset, returns the byte within the associated file at that offset. We destruct the function in to three parts. First, we check if the byte requested is within the file.

```
Definition fetchInodeByte (disk: Disk) (superblock: SuperBlock)
  (inode: Inode) (bytePos: Z): Exc Z :=
  if inode.(size) <=? bytePos then
    error
  else
    ...
```

Next, we determine the block address using the recursive function we described above. Depending on the byte position requested, the block address which of the data block which will contain it may be either directly addressable (in which case we grab the address from the Inode's list), or we may need to use one of the indirection blocks.

```
    ...
    let blockSize := (blockSize superblock) in
    let blockNumInFile := bytePos / blockSize in
    let directAddressable := 12 in
    let indirect1Addressable := blockSize / 4 in
    let indirect2Addressable := (blockSize * blockSize) / 16 in

  (* Requested byte makes sense? *)
  if inode.(size) <=? bytePos then
    error
  else
    (* Determine Block Address *)
    (if blockNumInFile <? directAddressable then
      nth_error inode.(block) (Z.to_nat blockNumInFile)

     else if blockNumInFile <? directAddressable
                              + indirect1Addressable then
      (nth_error inode.(block) 12)
        _flatmap_ (fun indirectBlock ⇒
        walkIndirection disk superblock
          (blockNumInFile - directAddressable)
          (ba2Offset superblock indirectBlock)
          0
        )

    else if blockNumInFile <? directAddressable
                             + indirect1Addressable
                             + indirect2Addressable then
      (nth_error inode.(block) 13)
        _flatmap_ (fun doubleIndirectBlock ⇒
        walkIndirection disk superblock
          (blockNumInFile - directAddressable
                          - indirect1Addressable)
          (ba2Offset superblock doubleIndirectBlock)
          (S 0)
        )

    else (nth_error inode.(block) 14)
      _flatmap_ (fun tripleIndirectBlock ⇒
        walkIndirection disk superblock
          (blockNumInFile - directAddressable
                          - indirect1Addressable
                          - indirect2Addressable)
          (ba2Offset superblock tripleIndirectBlock)
          (S (S 0))
      )
```

The steps needed to find a given byte within a file, then, are to determine the block address of the block which contains that byte (perhaps walking

```
indirection blocks to get there), and then jumping to the appropriate byte
within that data block. The {\tt fetchInodeByte} function therefore checks if
the requested byte is valid, determines which data block the byte is in, and
then pulling the associated byte out of that block.

\begin{lstlisting}
Definition fetchInodeByte (disk: Disk) (superblock: SuperBlock)
  (inode: Inode) (bytePos: Z): Exc Z :=
  let blockSize := (blockSize superblock) in
  let blockNumInFile := bytePos / blockSize in
  let directAddressable := 12 in
  let indirect1Addressable := blockSize / 4 in
  let indirect2Addressable := (blockSize * blockSize) / 16 in

  (* Requested byte makes sense? *)
  if inode.(size) <=? bytePos then
    error
  else
    (* Determine Block Address *)
    (if blockNumInFile <? directAddressable then
        nth_error inode.(block) (Z.to_nat blockNumInFile)

     else if blockNumInFile <? directAddressable
                              + indirect1Addressable then
        (nth_error inode.(block) 12) _flatmap_
          (fun indirectBlock ⇒
            walkIndirection disk superblock
              (blockNumInFile - directAddressable)
              (ba2Offset superblock indirectBlock)
              0
          )

    else if blockNumInFile <? directAddressable
                             + indirect1Addressable
                             + indirect2Addressable then
        (nth_error inode.(block) 13) _flatmap_
          (fun doubleIndirectBlock ⇒
            walkIndirection disk superblock
              (blockNumInFile - directAddressable
                              - indirect1Addressable)
              (ba2Offset superblock doubleIndirectBlock)
              (S 0)
          )

    else (nth_error inode.(block) 14) _flatmap_
      (fun tripleIndirectBlock ⇒
          walkIndirection disk superblock
            (blockNumInFile - directAddressable
                            - indirect1Addressable
                            - indirect2Addressable)
            (ba2Offset superblock tripleIndirectBlock)
            (S (S 0))
        )

    ) _flatmap_ (fun blockAddress ⇒
      (* With the block address, get the individual byte *)
      disk (blockSize * blockAddress + (bytePos mod blockSize))
    ).
```

5.4 Parsing File Names from a Tar

Once we've been given the unzipped tar file (i.e. from `gunzip_a`) we need to parse its structure well enough to pull out the file names it contains. This is not terribly dissimilar to pulling out SuperBlocks, Inodes, etc. from a Disk, save that here we are working within the context of a single file. As files are abstracted to the point that they need only provide a function to access their data, we need not worry about underlying file system mechanics.

A tar file is composed of a sequence of (header, file content) pairs such that the header meta data about the file (including its name, size, owner, etc.) and the file data is padded to a multiple of 512 bytes. To retrieve all of the file names contained within a tar, we will need to read the next file name from the tar header, read the file size from that header, skip the file contents and start again with the next header.

5.4.1 File Size from ASCII Octal

File size can be determined by reading the 11 bytes starting at offset 124. The size is first encoded into octal, which is then represented as ASCII characters. Scratching your head yet? We implement this with a simple recursive function which reads the next byte from a list, converts that into the encoded integer value. The value is multiplied by 8 raised to the length of rest of the list to account for octal order of magnitude. If the byte does not fall into the range associated with ascii '0' through '7', an `error` is returned.

```
Fixpoint fromOctalAscii (bytes: list Z) : Exc Z :=
  match bytes with
  | nil ⇒ value 0
  | byte :: tail ⇒ match (fromOctalAscii tail) with
    | error ⇒ error
    | value rest ⇒ if (andb (48 <=? byte) (byte <=? 56))
      then value (rest + ((byte-48)
                          * (8 ^ (Z.of_nat (length tail)))))
      else error (* Invalid character *)
    end
  end.
```

5.4.2 More Functional Idioms

The first hundred bytes of the header are dedicated to the filename (regardless of filename length); the filename is null-terminated. To get there, we will add three new functional idioms: `upto`, `flatten`, and `takeWhile`. `upto` (or `range`) acts as a simple, infix way to create a sequence of integers between the provided end points; we skip its definition as it involves too much Coq-specific inside baseball. Similar to our need for `_flatmap_`, `flatten` takes a list of options and converts them to a list of the options' contents. Finally `takeWhile` is a function that takes two parameters, a boolean predicate and a list of elements. This function returns the longest prefix of that list such that every element satisfies the predicate.

```
Fixpoint takeWhile {A} (predicate: A→ bool) (lst: list A)
  : list A :=
  match lst with
  | head :: tail ⇒ if (predicate head)
                   then (head
                         :: (takeWhile predicate tail))
                   else nil
  | nil ⇒ nil
end.

Fixpoint flatten {A} (lst: list (Exc A)): list A :=
  match lst with
    | (value head) :: tail ⇒ head :: (flatten tail)
    | error :: tail ⇒ flatten tail
    | nil ⇒ nil
  end.
```

5.4.3 Parsing a Single Header

We want to provide a function which can both pull out the first filename from a tar file as well as create the associated file. As our concept of a file is relatively abstract, we need only parse the file size (as described above), carry over the deleted status of the parent tar file, and drop the header from the data to retrieve the first file.

```
Definition parseFirstFileNameAndFile (tar: File)
  : Exc (ByteString*File) :=
  let firstHundredBytes := map tar.(data) (0 upto 100) in
  let fileName := flatten (takeWhile
    (fun (byte: Exc Z) ⇒ match byte with
      | error ⇒ false
      | value 0 ⇒ false
      | value _ ⇒ true
    end) firstHundredBytes) in
  (seq_list tar.(data) 124 11)
    _flatmap_ (fun fileSizeList ⇒
  (fromOctalAscii fileSizeList) _map_ (fun fileSize ⇒
    (fileName,
     (mkFile fileSize
```

```
            (* Keep the deleted status of the tar *)
            tar.(deleted)
            (* Header size = 512 *)
            (shift tar.(data) 512)))
  )).
```

5.4.4 Recursing Through

Now that we have a function which fetches the first filename and file from the tar, we can call it recursively, creating a new version of the tar file (i.e. minus the first file) with each step. This is relatively straight forward save that we will need to pad the file contents to 512 bytes. We won't use a Fixpoint but opt for a parameter to signify the recursive call; this is required by `N.peano_rect`, a library function which allows for recursive calls over binary numbers. To learn more, please see the source code.

```
Definition recFileNameFrom (nextCall: File → list ByteString)
  (remaining: File) : list ByteString :=
  if (remaining.(fileSize) <=? 0)
    then nil
  else match (parseFirstFileNameAndFile remaining) with
    | error ⇒ nil
    | value (fileName, file) ⇒
        (* Strip the first file out of the tar *)
        (* Round to the nearest 512 *)
        let firstFileSize := (
          if (file.(fileSize) mod 512 =? 0)
          then file.(fileSize) + 512
          else 512 * (2 + (file.(fileSize) / 512))) in
        let trimmedTar :=
          (mkFile (remaining.(fileSize) - firstFileSize)
                  remaining.(deleted) (* Parent's value *)
                  (shift remaining.(data) firstFileSize)) in
        fileName :: (nextCall trimmedTar)
  end.
```

6. Timelines as Evidence

The second type of evidence provided by the Honeynet researchers was in the form of a timeline of events to explain what they believed happened to the infected server. A timeline is simply an ordered sequence of events, and examples of events include file modification, user login, system restart, etc. Timelines are certainly useful as a form of evidence as they provide a narrative of what took place, but they are also provable artifacts, as a timeline is only sound as long as there is evidence for each of the events and if the order of those events can be verified.

```
Definition isSound (timeline: Timeline) (disk: Disk) :=
  (forall (event: Event),
    (* Event is evident from the disk *)
    (In event timeline) → (foundOn event disk))
  (* Events are in the correct sequence *)
  ∧ (forall (index: nat),
       (index < ((length timeline) - 1) )%nat →
         match (nth_error timeline index,
                nth_error timeline (index + 1)) with
         | (value lhsEvent, value rhsEvent) ⇒
           beforeOrConcurrent lhsEvent rhsEvent
         | _ ⇒ False
         end)
  .
```

6.1 Events and Their Relations

We next consider an `Event` type and its various forms. For files, we will see four events: access, creation, modification, and deletion. We will represent these events with two parameters, a unix-style timestamp of the event's execution and a unique identifier for the file (e.g. Inode Index in Ext2).

```
Inductive Event: Type :=
  | FileAccess: Z → Z → Event
  | FileModification: Z → Z → Event
  | FileCreation: Z → Z → Event
  | FileDeletion: Z → Z → Event
  (*| Execution: ByteString → Event *).
```

While not all events have a timestamp, those that do make the `beforeOrConcurrent` definition significantly simpler. To prove that one event happens `beforeOrConcurrent` another, simply compare the timestamps.

```
Definition timestampOf (event: Event) : Exc Z :=
  match event with
  | FileAccess timestamp _ ⇒ value timestamp
  | FileModification timestamp _ ⇒ value timestamp
  | FileCreation timestamp _ ⇒ value timestamp
  | FileDeletion timestamp _ ⇒ value timestamp
  | _ ⇒ error.

Definition beforeOrConcurrent (lhs rhs: Event) :=
  match (timestampOf lhs, timestampOf rhs) with
  | (value lhs_time, value rhs_time) ⇒ lhs_time <= rhs_time
  | _ ⇒ False.
```

Note that the `beforeOrConcurrent` relation need not be limited to Events where we have a concrete timestamp. We could extend the definition of an Event to include execution of shell scripts, for example, which have a clear relative ordering of timestamps but do not have absolute time.

We also define a function which pulls out the inode of an event (as this is a common operation amongst the four types of Events we have defined.) This is almost identical to the timestamp retrieval.

```
Definition inodeIndexOf (event: Event) : Exc Z :=
  match event with
  | FileAccess _ inodeIndex ⇒ value inodeIndex
  | FileModification _ inodeIndex ⇒ value inodeIndex
  | FileCreation _ inodeIndex ⇒ value inodeIndex
  | FileDeletion _ inodeIndex ⇒ value inodeIndex
  end.
```

6.2 Finding Events and Their Existence

For a Timeline to be valid, each of the events in the timeline must follow from the disk image. As the events we have discussed so far have an associated Inode, it's easy to compute the Inode structure and verify that its access, modification, change, or deletion time match that of the event.

```
Definition foundOn (event: Event) (disk: Disk) : Prop :=
  match (inodeIndexOf event) _flatmap_ (fun inodeIndex ⇒
    (ext2_inode_from disk inodeIndex) _map_ (fun inode ⇒
      match event with
      | FileAccess timestamp _ ⇒
        inode.(accessTime) = timestamp
      | FileModification timestamp _ ⇒
        inode.(modificationTime) = timestamp
      | FileCreation timestamp _ ⇒
        inode.(changeTime) = timestamp
      | FileDeletion timestamp _ ⇒
        inode.(deletionTime) = timestamp
      end
    )
  ) with
  | error ⇒ False
  | value prop ⇒ prop
  end.
```

6.3 Applying to Honeynet

With all of these definitions we can now see how one would provide evidence that a particular timeline was valid. Consider Jason Lee's entry[7] into the Honeynet contest described before. As part of his evidence, he provided a sequenced list of inode events (as discovered by "MACtimes") and annotated their significance. We copy several of these events and their annotations into Coq, losing only the file name (instead using inode number).

```
Lemma lee_honeynet_file:
  (Timeline.isSound (
    (* Mar 16 01 12:36:48 *)
      (* rootkit lk.tar.gz downloaded *)
        (FileModification 984706608 23)
    (* Mar 16 01 12:44:50 *)
      (* Gunzip and Untar rootkit lk.tar.gz *)
        :: (FileAccess 984707090 23)
      (* change ownership of rootkit files to root.root *)
        :: (FileAccess 984707102 30130)
      (* deletion of original /bin/netstat *)
        :: (FileDeletion 984707102 30188)
      (* insertion of trojan netstat *)
        :: (FileCreation 984707102 2056)
      (* deletion of original /bin/ps *)
        :: (FileDeletion 984707102 30191)
      (* insertion of trojan ps *)
        :: (FileCreation 984707102 2055)
      (* deletion of origin /sbin/ifconfig *)
        :: (FileDeletion 984707102 48284)
      (* insertion of trojan ifconfig *)
        :: (FileCreation 984707102 2057)
    (* Mar 16 01 12:45:03 *)
      (* the copy of service files to /etc *)
        :: (FileAccess 984707103 30131)
      (* hackers services file copied on top of original *)
        :: (FileCreation 984707103 26121)
    (* Mar 16 01 12:45:05 *)
      (* deletion of rootkit lk.tar.gz *)
        :: (FileDeletion 984707105 23)
    :: nil)
    honeynet_image_a
  ).
```

7. Future Work

Anyone not already familiar with Coq's tactic systems would be quite confused by the steps needed to prove any of these lemmas. On the other hand, those who are familiar with Coq might find the compute-heavy nature of these definitions to be outside of their comfort zone. Compound that with the rather large numbers needed, and it becomes safe to say that very few people would be able to use this system, and none work in the professional forensics space.

A potential solution appears in two phases. First, these proofs are ripe for automation. Providing one of a pre-defined set of evidence types and a disk image should be enough for a script to generate the required assumptions, evidence, and proof terms. As a second step, we could develop a meta language or other interface for describing the types of evidence needed and allow a program to search and generate proofs as needed. This connects with wider work by Radha Jagadeesan, Corin Pitcher, and James Riely of DePaul University, which includes efforts to automate forensic methods.

This paper describes only a few of the types of evidence that would be required for a complete forensics tool. We used a single honeynet challenge to provide this paper scope. Obviously, creating additional definitions of evidence would be a key area for future research. This means both tasks such as defining additional file systems and describing additional types of events. Imagine what types of events would describe a user logging in, executing a command, and then disconnecting. Imagine the evidence needed to prove that a user frequently visited a certain domain.

Finally, the vast majority of this paper has been devotes to computations or definitions; we have not described many relations between the definitions. It would be worthwhile to proof propositions about these definition (as we proved that JPEG files cannot be Tars). We might prove that a patch applied to a disk image could restore a deleted file (as studied by Charles Winebrinner), that deletion events imply the associated file is deleted, that files can exist which are not addressable, or any number of other proofs and lemmas. These would then make proofs about particular disk images even easier to apply.

Acknowledgments

First, I wish to thank my colleagues at DePaul, including Malik Aldubayan, Iana Boneva, Christina Ionides, Daria Manukian, Matthew McDonald, and Charles Winebrinner for their ideas and community. I would also like to thank my professors, Radha Jagadeesan, Corin Pitcher, and James Riely for their feedback and insights. Most notably, working with my advisor, Corin, has been a great pleasure. Most of the code described comes from kernels he provided, and I certainly would still be debugging Gecode if he hadn't stepped me through dozens of proofs. Thank you, all.

References

[1] The Honeynet Project. http://www.honeynet.org/

[2] The Honeynet Project *Scan of the Month* #15. http://old.honeynet.org/scans/scan15/

[3] Borland, Matt. Submission to Honeynet.org *Scan of the Month*, 05/05/2001. http://old.honeynet.org/scans/scan15/som/som6.txt

[4] Sleuth Kit

[5] http://www.nongnu.org/ext2-doc/ext2.html

[6] http://ofps.oreilly.com/titles/9780596155957/FunctionalProgramming.html

[7] http://old.honeynet.org/scans/scan15/som/som33.html#7