Formalizing the Honeynet

Defining and Proving a Rootkit Installation

CM Lubinski
DePaul University
cm.lubinski@gmail.com

Abstract

Formal definitions of multiple file formats, "rootkit-like" tar files, and event timelines are provided in the proof-oriented language, Coq. These are combined to mimic the types of evidence given by independent forensics researchers in a forensics competition ("Honeynet".) Using said definitions, the evidence is proven to be consistent with the disk image provided for the relevant competition. Along the way, functions for parsing and manipulating relevant data structures within Coq are also described.

Keywords forensics, formalization, Coq, Honeynet, Ext2

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 suspect that malicious software has been installed by the country's authorities, but are uncertain how to prove this. At first, you consider using off-the-shelf antivirus software to find the infection, but realize that the risk of damaging international relations is far too great to rely on the definitions within the proprietary software. What you really need is a concrete, standardized definition of what it means to have malware installed and evidence fitting that definition to prove that your machine has been invaded.

This scenario and others like it (though, perhaps more mundane) are the ultimate goal of the research provided in this paper. To achieve that, we need formal definitions for various types of evidence used by forensics analysts. We also require a way to convert real data (e.g. from a disk image) into these definitions such that we can provide concrete *proofs* showing that a particular image does or does not fit such a definition. In the above example, we could provide a verifiable proof that the disk image satisfied a definition of "malware installation".

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[2]'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, participants were challenged to describe what happened to the system and provide evidence for their conclusions. We will consider one specific contest[3], in which a rootkit (replacement system programs that hide malicious activity) was installed on a server. 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.

We will describe our definitions, code our parsers, and implement our evidence in the proof-oriented language, Coq. While this language lives in the ML-family of programming languages, we will augment it with several functional and object-oriented notations. Using Coq allows us to formalize our proofs and definitions, but significantly restricts the practicality of our efforts; we will expand on these limitations throughout the paper.

By the end of this write-up, we will have provided definitions used by some of the Honeynet contestants. These include definitions of tar files, rootkit-like archives, file deletion (on the Ext2 file system,) as well as a timeline of events which would be consistent with rootkit installation. Given portions of the same disk provided to those researchers, we will also present proofs that satisfy these definitions.

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 need only 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; most applications are lenient with the file formats they accept and messy with the files they write.

Instead, we will chose a middle route, opting to use "magic numbers" as our guide. This refers to tell-tale byte values at predictable offsets 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 Coq (plus some Python-like 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 unsigned byte values shown. We will discuss the absence of byte values later, but for now, assume that this accounts for "missing" evidence which might arise if a disk were damaged or inconsistent. By defining file types in this manner, we can use these definition as building blocks within larger definitions and within proofs. For example, we can now *prove* that JPEG files cannot also be gzipped files:

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

The tactics used to satisfy this proof (and others mentioned throughout this paper) can be found in the code referenced in the appendix.

3. Bird's Eye View of a Honeynet Proof

Let's now consider the Honeynet *Scan of the Month* mentioned in the introduction. In his entry for this contest, Matt Borland[1] 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
  ∧ Tar.looksLikeRootkit (gunzip_a file).
```

That's a bit of a mouthful; we are stating that there exists a deleted, gzipped file on the Honeynet disk image such that, when we unzip that file, the contained tar looks like a rootkit. For the remainder of this section, we will dive deeper into each of the definitions in the "and" clause (save `isGzip`, which we have already described.) Section 5 will investigate this evidence even closer, and include a description of the assumptions (as indicated by `_a`.)

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 on 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, disjoint segments of a disk image. Moreover, this definition has high potential for false positives, where a "file" is formed by looking at the bits that span a fragment boundary but are not part of the same byte sequence. Files are stored on disks via file systems, which make no sequential guarantees (particularly in recent file systems such as ZFS and Btrfs.) On spinning hard drives, the need to "defragment" arises from the segmentation of files over non-adjacent sectors on disk; moving the fragments to be adjacent improves read performance by reducing the times the drive head must skip around.

As sequential access will not do, we must 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, so we will define file existence as the disjunction of file existence within each file system. While file systems are distinct, they each tend to use numeric file identifiers for each file on the system. Hence, for most file systems, a file exists if we can provide an identifier that maps to it.

```
Definition isOnDisk (file: File) (disk: Disk):
  (* Ext2 *)
  (exists (inodeIndex: Z):
    fileEq (Ext2.findAndParseFile disk inodeIndex)
           (value file))
  ∨ (* FAT32 *)
  (exists (clusterNumber: Z):
    fileEq (Fat32.findAndParseFile disk clusterNumber)
           (value file))
  ∨ (* Btrfs *)
  (exists (key: Z):
    fileEq (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 and `fileEq` in the next section.

3.2 The File Structure and 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 or marked free. To account for these varieties, we will include the deletion status of a File within its *definition*, allowing each file system to define the status how it wishes. This definition provides three "universal" fields and five "optional" fields.

```
Structure File := mkFile {
  fileSystemId: Exc Z;
  fileSize: Z;
  deleted: bool;
  byteOffset: ByteData;
  lastAccess: Exc Z;
  lastModification: Exc Z;
  lastCreated: Exc Z;
  lastDeleted: Exc Z
}.
```

The first field is an identifier specific to the file system that contains the file, e.g. the Inode Index in Ext2. This, along with the `last*` fields might not be present in every file or in every file system, but provide a great deal of information when they are. We will use these fields as the basis for our Timeline Events at the end of this paper. As the fields might not be present for each file system, they are marked as optional (indicated by `Exc`, described further in section 5.1.)

The `fileSize` and `deleted` fields are self-explanatory, but `byteOffset` warrants additional review. This field is a function that, when given an offset within a file, returns the byte value found at that offset within the file *data*. 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,) so we must wrap the result in an Exc (a.k.a. Option.)

```
Definition ByteData := Z → Exc Z.
```

Given the definition of Files, we can describe `fileEq`, our first work-around required by our language choice. We cannot rely on native equality because we want the `byteOffset` field to be flexible enough to allow for complex, partially applied functions. While we want to define equality of such functions as pairwise mappings between input and output, Coq would attempt to normalize them before comparing. The functions we will use are too large (when normalized) for Coq to handle in a reasonable time frame. Instead, we will use a custom relationship, `fileEq`, which operates on two

`Exc Files`, and holds if the simple fields match and if, for all offsets, the bytes returned by `byteOffset` match.

```
Definition fileEq (lhs rhs: Exc File) :=
  match (lhs, rhs) with
  | (error, error) ⇒ True
  | (value lhs, value rhs) ⇒
      lhs.(fileSize) = rhs.(fileSize)
      ∧ lhs.(deleted) = rhs.(deleted)
      ∧ forall idx:Z, lhs @[ idx ] = rhs @[ idx ]
  | _ ⇒ False
  end.
```

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 Tars That Look Like Rootkits

The final clause in our lemma relies on the `gunzip_a` assumption, which we will describe shortly; for now, know that this is a representation of a decompression algorithm for the gzipped files. If we can prove that the result of unzipping the file satisfies the `looksLikeRootkit` definition, we have finished our proof.

If we peal this definition back a layer, we will see that it requires there to be two distinct file names (represented as lists of bytes) that are are in the list of file names contained in that tar (`parseFileNames`) such that the file names are `systemFiles`. We require *two* file names as this increases the odds that the tar is malicious (rather than a legitimate update.)

```
Definition ByteString := list Z.

Definition looksLikeRootkit (file: File) :=
  exists (filename1 filename2: ByteString),
    (In filename1 (parseFileNames file))
    ∧ (In filename2 (parseFileNames file))
    ∧ (FileNames.systemFile filename1)
    ∧ (FileNames.systemFile filename2)
    ∧ filename1 <> filename2.
```

System file names are distinguished from other files in that they (excluding their path prefixes) are in a predefined list. This list includes task managers, such as `top` and `ProcMon.exe`, which rootkits replace to hide their activities, as well as `ssh` and `rsync`, which grants them the ability to monitor network traffic.

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

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. For the ease of description, however, we write system file names in ASCII, and convert them all into ByteStrings via the `ascii2Bytes` function, described in section 5.5

Use of this definition is straight forward:

```
Lemma systemFile_ex1 :
  systemFile (ascii2Bytes "last/top").

Lemma systemFile_ex2 :
  ¬(systemFile (ascii2Bytes "last/example.txt")).
```

3.4 Summary

We have now described (at a high level) `isOnDisk`, `isDeleted`, `isGzip`, and `looksLikeRootkit` enough that we could see why Borland proposed them as definitions of evidence for a rootkit installation. This is not complete, however, as we skipped over several critical parsing and computation functions which we will expand on in section 5. Before describing these functions, we should clarify the relationships between some of the concepts we are proposing; this is the focus of the next section.

4. Lemmas, Computations, 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 forensics concept. For example, we provided definitions for `isGzip`, `looksLikeRootkit`, etc. While one of the wider goals of this research is to provide a set of such common definitions, this paper only describes several which are relevant to the Honeynet example. 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." Note that definitions are, at their core, the choice of their creators; forming a consensus on a definition is the only way it may provide authority. This also means that the possibility for multiple definitions is present (as we have shown with our two definitions for file existence.)

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 separating particular parsing computations (which build the structures) from definitions involving the structures. While not complete, this division allows us to reason about our definitions using only the data representations, not the parsing algorithms. It should be relatively simple for someone to write a different parser which produces the same structures without affecting many of the definitions.

Due to Coq's proof-oriented nature, we have found several limitations that need to be worked around in our implementation. Coq programs cannot, for example, read bytes from a disk image; they cannot (for our intents) call external programs to transform data; they 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 full disk image; 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 an unzipping function – they instead trust those assumptions to be accurate and rely on users for verification. We will describe the assumptions made for our Honeynet example in detail in section 5.2.

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, reading file names from a tar archive, generating modify-access-create times for Inodes, or creating "subtars" by striping off the first file's information from an archive. 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 have described required researchers 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 that type of evidence for the attacked server. This led each researcher to give different forms of evidence depending on what that researcher found to be acceptable. The types of evidence provided are codified in our definitions (as described above,) but the evidence they gave maps closer to a Coq **lemma** or proof. The evidence will be in the form of a proof term, which the author of a proof can build up using Coq's tactic system. Given a proof that the disk satisfies the definitions provided, it is only up to a reviewer to determine whether or not the *definitions* are satisfactory. We provide a list of tactics that generate the proof term for each of our lemmas in the source linked to in our appendix, and we will expand the implications of this topic further, in section 7.

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

Now that we have introduced these concepts, let's step through our `borland_honeynet_file` proof, which will touch on each.

5.1 A Note on Functional Idioms

While parsing, we will make use of several idioms from functional programming. In particular, we make heavy use of the "option" pattern; due to the sparse map that we use to represent the disk, the parsing code does not know whether requested bytes will be present. To account for this fact, each attempt to read returns an *option* of the result. An option is simply a wrapper around a value such that the wrapper may 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`, indicating an option of type `A`.

The benefit of wrapping the value in the monadic option is that we can continue computation without knowing (in advance) whether a sub-computation was successful or not. Options make error handling "percolate up" in that functions which produce options can be chained such that *any* error in a component causes the entire computation to result in an error. To get to this point, we 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_` is to transform the contents of an option, if present. If the option is empty, map has no effect. `_flatmap_` similarly does not affect empty options. However, if a value is present, the provided function (`fn`) is applied and the option is replaced with that function's result. In other words, `_flatmap_` is like applying a `_map_` and then stripping the outer `Exc`. These functions allow 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 (Z, Z, Z)` (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 have rushed through these concepts as we assume the reader has some background in functional programming. For a more thorough (yet still brief and practical) introduction, see Wamper & Miller[7].

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 `value 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. We leave a full accounting of the disk bytes to our code (see the comments in `example_images.v` for an explanation of why each byte range is included.) Note: `find` is an operation to reach into a `FMapAVL`.

```
Definition honeynet_map :=
[
  1024 |→ 216,
  1025 |→ 2,
  1026 |→ 1,
  1027 |→ 0,
  ...
].

Definition Disk_of_Map_Z_Z (map: Map_Z_Z) : Disk :=
  fun (key: Z) ⇒ find key map.

Definition honeynet_image_a : Disk :=
  Disk_of_Map_Z_Z honeynet_map.
```

The second assumption (`gunzip_a`) represents gzip's unzipping/deflating operation. As implementing the decompression algorithm within Coq would not be particularly useful to our study, 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) into a `FMapAVL`. We effectively just wrap this data in a function that, regardless of `File` input, returns the uncompressed file data.

```
Definition gunzipped_23 :=
[
  0 |→ 108,
  1 |→ 97,
  2 |→ 115,
  ...
].

Definition gunzip_a := (fun (input: File) ⇒
  (mkFile None (* no need for an id *)
          1454080 (* uncompressed file size *)
          input.(deleted)
          (fun (offset: Z) ⇒ find offset gunzipped_23)
          (* Fields not used; ignore them *)
          None None None None)).
```

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 data structures created.

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 some file-system-specific fields, a file size, deletion status, and a function that retrieves bytes within the file. These fields require we first parse out the `SuperBlock` of the disk as well as the `GroupDescriptor` and `Inode` associated with our Inode index (`23` in the above example.)

To understand what these structures represent, 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 contiguous (or even in-order) 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 collective information about Inodes, such as which are allocated as well as provides a mechanism to segment the administration of Inodes. We can easily compute to which group a particular Inode index belongs by dividing it by the number of inodes per group (a file-system-wide setting found in the SuperBlock.) We can also compute the Inode's position within that group by taking the remainder of this division (i.e. by applying mod.) It's important to note for both of these operations that Inode indices are one-indexed; this will lead to some additions and subtractions by one throughout our code to convert between zero- (with which it is easier to compute) 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 some of the redundant 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)) 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
      (value inodeIndex)
      inode.(size)
      deleted
      (fetchInodeByte disk superblock inode)
      (value inode.(atime))
      (value inode.(mtime))
      (value inode.(ctime))
      (value inode.(dtime))
  )))).
```

Stripping off this layer, we next take a look at each of the composing computations, `findAndParseSuperBlock`, `findAndParseGroupDescriptor`, `findAndParseInode`, and `parseDeleted`, as well as the `fetchInodeByte` function, which has been partially applied.

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 requesting bytes. This wrapper allows us to parse as if the 1024th index were at position zero.

```
Definition shift (bytes: ByteData) (shiftAmount index: Z)
  : Exc Z :=
  bytes (shiftAmount + index).
```

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 Coq's `Z` type, 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.

With that, we can show our SuperBlock-parsing function. 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 ⇒
  (* ... Additional fields omitted ... *)
  (seq_lendu disk 260 4) _map_ (fun firstMetaBg ⇒
    mkSuperBlock
      inodesCount
      blocksCount
      (* ... 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. That field therefore 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 spans on disk. This is encoded using the logarithmic scale, but we provide a simple function to convert to the base-10 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. 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, which is encoded in the SuperBlock as described above. With that, we can treat the disk as an array of blocks and simply jump to the relevant byte position.

```
Definition BA := Z.

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. If the block size is greater than 1024 (and hence, the SuperBlock is part of block zero,) the GroupDescriptors begin at block one. Otherwise, the SuperBlock composes all of block one, so the GroupDescriptors can be found at block two.

GroupDescriptors will be represented by a structure similar to SuperBlocks, though with far fewer fields. GroupDescriptors have 32 allocated bytes, yet 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. As with the SuperBlock, we will shift the disk to aid our parsing efforts; unlike the SuperBlock, GroupDescriptors are small enough that we will include their full parsing code here.

```
Definition findAndParseGroupDescriptor
  (disk: Disk) (superblock: SuperBlock) (groupId: Z)
  : Exc GroupDescriptor :=
  let groupBlockArrayBA := if (blockSize superblock >=? 1024)
    then 1 else 2 in
  let groupBlockArrayOffset :=
    ba2Offset superblock groupBlockArrayBA in
  let descriptorOffset := 32 * groupId  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 a block address pointing to the start of a sequence of inode-related data. The former is simply a bit sequence 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. As mentioned before, Inode indices are one-indexed, so the first Inode is associated with index 1.

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 – we know that Inodes are 128-byte data structures. 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 ⇒
    (* ... Additional fields omitted ... *)
    (seq_lendu disk 40 4) _flatmap_ (fun directBlock1 ⇒
    (* ... 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
        (* ... Additional fields omitted ... *)
        (directBlock1 :: directBlock2 :: directBlock3
          :: directBlock4 :: directBlock5 :: directBlock6
          :: directBlock7 :: directBlock8 :: directBlock9
          :: directBlock10 :: directBlock11 :: directBlock12
          :: indirectBlock :: doubleIndirectBlock
          :: tripleIndirectBlock :: nil)
        (* ... Additional fields omitted ... *)
        osd2
    ))))))))))))))))))))))))))))))))).
```

This code sample highlights the twelve "direct block" pointers and three indirection pointers, which we combine into a list, making the `block` field (keeping with Poirier's conventions.) 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.

5.3.6 parseDeleted

With the Inode and its index, we know the File's file system identifier, size, and modification-access-creation-deletion (MAC) times. We next turn to the field indicating whether or not the file is marked as deleted, or "unallocated." To find the allocation status of an Inode, we will need to find the associated allocation bitmap. As mentioned above, the bitmap associated with a particular Inode is indicated in that Inode's GroupDescriptor. As we are only reading whole bytes at a time, we will read the byte that contains the allocation bit for the Inode we care about and then test the bit (1, allocated, translates to true; 0 to false) within that byte.

Ultimately, this means we will find the start of the allocation bitmap from the GroupDescriptor, jump to the byte within that bitmap, read it, and test it. 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

We need only one more attribute to create our File structure. File objects have a method which, given an arbitrary offset, 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.

Remembering the structure of Ext2 Inodes, we know that determining which data block a given byte of a file is located in may require walking one or more levels of indirection. 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.

```
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)
  ...
```

If the function has not reached its base case (as indicated by `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)
  ...
  | S nextIndirectionLevel ⇒
    (* Type conversion *)
    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; if not, we immediately `error`.

```
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 described above. We begin by noting bounds on which data blocks are addressable directly and at the levels of indirection.

```
Definition fetchInodeByte (disk: Disk) (superblock: SuperBlock)
    ...
    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
    ...
```

How we attain the block address of the data depends on the byte requested. If the byte is in the first 12 data blocks, we have immediate access to the block address from the Inode's `block` list.

```
Definition fetchInodeByte (disk: Disk) (superblock: SuperBlock)
    ...
    let blockAddress :=
    (if blockNumInFile <? directAddressable then
      nth_error inode.(block) (Z.to_nat blockNumInFile)
    ...
```

If, instead, the block falls within the range of the first indirection block, we need to call our `walkIndirection` function.

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

If the block falls within the range of the double indirection block, we call `walkIndirection` with an additional level. Everything else falls to the triple indirection block.

```
Definition fetchInodeByte (disk: Disk) (superblock: SuperBlock)
    ...
    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 O)
    )
  else
    (nth_error inode.(block) 14)
      _flatmap_ (fun tripleIndirectBlock ⇒
      walkIndirection disk superblock
        (blockNumInFile - directAddressable
                        - indirect1Addressable
                        - indirect2Addressable)
        (ba2Offset superblock tripleIndirectBlock)
        (S (S O))
    )
  ) in
  ...
```

Finally, once we have the block address for the data block, we need to jump to the actual byte requested. We determine this simply by taking the byte position requested modulo the size of each data block.

```
Definition fetchInodeByte (disk: Disk) (superblock: SuperBlock)
  ...
  let blockAddress := (
    ...
  ) in
  blockAddress _flatmap_ (fun blockAddress ⇒
    disk ((ba2Offset superblock blockAddress)
          + (bytePos mod blockSize))
  ).
```

With this function, we have completed each of the components needed to represent a file structure from an Ext2 disk. Putting them together (via `mkFile`) allows us to use that file as the evidence needed for much of `borland_honeynet_file`. In particularly, we know how to prove that the file `isOnDisk` (if we can provide an Inode Index that matches it,) that it the file `isDeleted` (if the deleted flag is set,) and that the file `isGzip` (if the first three bytes are those of a Gzip.)

All that's left is to show that the decompressed tar `looksLikeRootkit`.

```
Lemma borland_honeynet_file:
  ...
  ∧ Tar.looksLikeRootkit (gunzip_a file).
```

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 contains 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 "first" file name from the tar header, read the file size from that header, skip the file contents and start again with the following header.

5.4.1 File Size from ASCII Octal

File size of a file within a tar can be determined by reading the 11 bytes starting at offset 124 of the tar header. The size is first encoded into octal, which is then represented as ASCII characters. Encoding in ASCII stems from the desire to keep the header human readable; the choice of octal is a bit more dubious, but *c'est la vie*. We implement our parsing with a simple recursive function which reads the next byte from a list and 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 as anything else does not fit our encoding expectations.

```
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, skipping any `errors`. 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 in the prefix satisfies the predicate.

```
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.

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.
```

5.4.3 Parsing a Single Header

Next, we will write a function that can both pull out the first filename from a tar as well as the first file's contents. Retrieving the file name requires reading the first hundred bytes of the tar and taking every character until a null is read. Due to our abstract concept of a file, extracting the first file from the tar is relatively straight forward. We need only parse the file size (making use of a sequence reader, `seq_list`,) carry over the deleted status of the parent tar file, and drop the tar header from the data to create a File object (we do not provide any of the optional values.)

```
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 (* stop *)
      | value 0 ⇒ false (* stop *)
      | value _ ⇒ true  (* continue *)
    end) firstHundredBytes) in
  (seq_list tar.(data) 124 11)
    _flatmap_ (fun fileSizeList ⇒
  (fromOctalAscii fileSizeList) _map_ (fun fileSize ⇒
    (fileName,
```

```
    (mkFile None (* no id in child files*)
            fileSize
            (* Keep the deleted status of the tar *)
            tar.(deleted)
            (* Header size = 512 *)
            (shift tar.(data) 512)
            (* Carry over fields from parent tar *)
            tar.(lastAccess)
            tar.(lastModification)
            tar.(lastCreated)
            tar.(lastDeleted)
  ))
)).
```

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. We need to strip off the initial file header (512 bytes) plus the file size padded to 512 bytes. We won't use a fixpoint, opting instead for a parameter (`nextCall`) to signify the recursive call; this is required by `N.peano_rect`, a library function which allows for recursive calls over binary numbers (including our `Z`.) To see the full definition of `Tar.parseFileNames`, 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 None (* No id in child files *)
                 (remaining.(fileSize) - firstFileSize)
                 remaining.(deleted) (* Parent's value *)
                 (shift remaining.(data) firstFileSize)
                 (* Use parent's values *)
                 remaining.(lastAccess)
                 remaining.(lastModification)
                 remaining.(lastCreated)
                 remaining.(lastDeleted)
         ) in
       fileName :: (nextCall trimmedTar)
  end.
```

5.5 Computing Over File Names

Now that we have pulled the list of file names from the tar file, we want the ability to prove that two, distinct file names satisfy `systemFile`. As we discussed before, this means that we want to compare the file name (sans directory) with an a priori set of known system files. As a proof of concept, the list provided is slim, but it would be quite simple to extend it.

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

The `ascii2Bytes` function simply converts each Coq ASCII character into a corresponding `Z` to represent it as a byte. We use the functional `map` idiom to run this function an every element of the list of Coq strings.

```
Fixpoint ascii2Bytes (fileName: string): ByteString :=
  match fileName with
  | EmptyString ⇒ nil
  | String char tail ⇒
    (Z.of_N (N_of_ascii char)) :: (ascii2Bytes tail)
  end.
```

We also need to define the `trimFileNamePrefix` function, which we want to return the string after the last directory delimiter (i.e. after the last '/' or '\'.) To do this, we reuse our `takeWhile` function applied to the reverse of the file name provided, taking bytes until we hit either '/' or '\'. We then reverse that result to get the final, trimmed file name.

```
Definition trimFileNamePrefix (fileName: ByteString)
  : ByteString :=
  let reversedName := rev fileName in
  rev (takeWhile
        (fun (char: Z) ⇒
          (negb (orb (char =? 47)    (* '/' *)
                     (char =? 92)))) (* '\' *)
        reversedName).
```

With that, we have described all of the components needed to compute our lemma. We leave the details of the proof to our code samples, but at the high level, one can imagine how proving `borland_honeynet_file` need only require we provide a File (by parsing it from the disk) as evidence. We can more or less compute each of the properties based on that file.

6. Timelines as Evidence

A second type of evidence provided by the Honeynet researchers (particularly from Jason Lee[4]) had the form of a timeline of events to explain what the researchers believed happened to the infected server. A timeline is simply an ordered sequence of events; 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. They are also provable artifacts, as a timeline is only sound as long as there is evidence for each of its events and if the order of those events can be verified. We consider events to be in the correct sequence if events earlier are `beforeOrConcurrent` events later in the sequence.

```
Definition isInOrder (timeline: Timeline) :=
  (* 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).

Definition isSound (timeline: Timeline) (disk: Disk) :=
  (forall (event: Event),
    (* Event is evident from the disk *)
    (In event timeline) → (foundOn event disk))
  ∧ isInOrder timeline.
```

6.1 Events and Their Relations

We next consider an `Event` type in 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 file system identifier (as would be found in a File.)

```
Inductive Event: Type :=
  | FileAccess: Z → Z → Event
  | FileModification: Z → Z → Event
  | FileCreation: Z → Z → Event
  | FileDeletion: Z → Z → Event
```

While not all conceivable 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 commands but do not have absolute time.

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. While conceivable events may not be associated with files, the four we have defined all require the existence of a file that confirms their relevance. We simply check that there exists a file on disk that shares the same file system identifier and MAC time.

```
Definition foundOn (event: Event) (disk: Disk) : Prop :=
  exists (file: File),
    isOnDisk file disk
    ∧ (match event with
        | FileAccess timestamp fsId ⇒
            file.(fileSystemId) = value fsId
            ∧ file.(lastAccess) = value timestamp
        | FileModification timestamp fsId ⇒
            file.(fileSystemId) = value fsId
            ∧ file.(lastModification) = value timestamp
        | FileCreation timestamp fsId ⇒
            file.(fileSystemId) = value fsId
            ∧ file.(lastCreated) = value timestamp
        | FileDeletion timestamp fsId ⇒
            file.(fileSystemId) = value fsId
            ∧ file.(lastDeleted) = value timestamp
       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[4] 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; in the conversion we lose file name (instead, we use inode number) and pretty-printed dates (opting for unix time stamps.)

```
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
  ).
```

Seeing as the definition of `Timeline.isSound` is composed of two computable relations, proving this lemma boils down to a series of computations. The full list of tactics used to build up the proof is linked from the appendix.

7. Discussion: Are These Definitions Sufficient?

In this paper, we have provided several definitions for types of evidence that would be applicable to forensics researchers. It is important to take a step back and determine whether or not those definitions provide the evidence we need to prove that a root kit was installed on the disk in question.

First, we note that the timeline we provided (and verified) consists only of modification, file system identifiers, and timestamps. We rely on the annotations to provide context about which file the identifier is associated with; if we were to swap these annotations with different file names, we could construct a completely different scenario. Let us assume, however, that we can derive a filename from the file system identifier (and can therefore verify annotations) as this is a more interesting situation.

Making this assumption, are there scenarios where our definitions could be satisfied without a rootkit being installed? Let's start by ruling out possibilities. While downloading and installing non-system software satisfies many of the requirements (perhaps a gzipped-tar that is deleted after replacing some files,) we can rule this scenario out based on the name of the files replaced (i.e. non-system software installation wouldn't affect system utilities.) What about an update to a system program? Here we can rely on the clause within our "malicious-looking" check which requires *multiple* system file names.

The obvious next step would be to ask what would happen if we were performing a system upgrade, one that would affect multiple system files. Here we might argue that practical package managers would not include the changes in a single archive, but that argument is not as strong as we desire. It's conceivable that a hotfix to a predictable operating system, say OS X, might come as a single, gzipped tar archive which replaces multiple system files. By all accounts, this would trigger our definition, even though a rootkit had not actually been installed.

Here, the leap between our definition and its semantics causes us pain – scenarios like the one described *look* like rootkit installation. System files are replaced by those found within an archive which was deleted shortly after installation. We might try to eliminate these false positives by checking for code signatures or describing rootkit files by a tell-tale fingerprint, either within the executable's bytes or within its actions. That said, making this exception would simply move the goal post further down the field; we could surely find a legitimate counter-example even with the additional restrictions.

We will never be able to completely remove counter-examples as given enough iterations, a random number generator would eventually generate a disk that provides *exactly* the data we need. Clearly, no rootkit was installed on such a constructed disk, yet any definition we could conceive could be satisfied. Ultimately, the

definitions cannot guarantee their semantic meaning; in aggregate the definitions can only make such meaning more and more likely.

This, too, is the methodology found within the Honeynet competition; as there were no canonical definitions for what it means for a rootkit to be installed, each participant provided his or her own evidence which satisfied his or her own definition. This led them to describe slightly different attacks, even though they all shared the same disk image.

We did not set out to provide bullet-proof definitions for concepts such as deletion, file creation, etc. Instead, we wanted to model the same types of evidence currently put forth by forensics researchers. If a consensus can be reached regarding these definitions, we would use it, but at the moment we model only certain de facto standards.

8. 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. 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 constraints and 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 events would define a user logging in, executing a command, and then disconnecting. What kind of evidence is needed to prove that a user frequently visited a certain domain while web browsing?

Finally, the vast majority of this paper has been devoted to computations or definitions; we have not described many propositional relationships. It would be worthwhile to prove propositions about these definitions (similar to how we showed JPEG files cannot be Gzips.) 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.

Thanks and great appreciation also go to my partner, Laura Cathey, who has tolerated my absence far too long. I promise I won't start another project for at least a few days.

References

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

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

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

[4] Lee, Json. Submission to Honeynet.org *Scan of the Month*, 05/25/2001. `http://old.honeynet.org/scans/scan15/som/som33.html`

[5] Poirier, Dave. The Second Extended File System. `http://www.nongnu.org/ext2-doc/ext2.html`

[6] The Sleuth Kit. `http://www.sleuthkit.org/`

[7] Wampler, Dean and Payne, Alex (2008) Chapter 8. Functional Programming in Scala. *Programming Scala*. `http://ofps.oreilly.com/titles/9780596155957/FunctionalProgramming.html`

A. Code Samples

Complete code samples and Coq tactics are available on Github: `https://github.com/cmc333333/forensics-thesis-code`