Feasible Formal Forensics

Computational Proofs of a Honeynet Challenge

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 through the proof-by-reflection technique. The applicability of this approach is also theorized, in addition to the role it could play in the wider field.

Keywords forensics, formalization, Coq, Honeynet, Ext2, Tar, reflection

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 found in the proprietary tool. 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.

Solutions to 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 satisfy such a definition. In the above example, we could provide a verifiable proof that the laptop matched 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[5]’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[6], 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 detailing how the rootkit was found.

The data provided by these contestants needs to be independently verified, preferably automatically, via a separate tool chain. Taking the entrants’ claims at face value is not an acceptable solution, but neither is using their forensics programs, as the programs’ findings must *also* be verified. We will instead formalize some of the definitions and types of evidence provided by the researchers and create independently verifiable *proofs* that the disk image given satisfies the claims made.

Our definitions, parsers, and evidence will be described in the proof-oriented application (and language), Coq. This tool allows us to formalize our efforts, but significantly restricts their practicality; we will expand on these limitations throughout the paper. Despite these complications, Coq lets us build complete proof terms which serve as the final, verifiable proof that the researchers’ assertions were accurate.

We will argue that, due to their method of construction, the proof terms generated allow our solution to be *feasible* in the sense that the terms remain a constant size rather than growing with the disk image, file, etc. We also provide sample running times for verifying the proofs and describe their relation to the size of their input. Combining the specific evidence with documentation for the feasibility of this strategy should demonstrate the approach’s potential as a core tool for forensics investigators. Along the way, we will include definitions of tar files, rootkit-like archives, file deletion (on the Ext2 file system,) as well as a timeline of events consistent with rootkit installation.

We aim to show that this approach not only works (via the Honeynet examples) but may also be scalable to the needs of forensics researchers.

2. Getting Our Feet Wet with File Types

Let us 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 tactic (used by many operating systems) is to rely on the file extension – 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 specification and confirm that all of the meta data contained within a file is consistent with said that document. That approach runs the opposite end of the spectrum, requiring significantly more evidence. Further, what is generally thought as a JPEG file is most likely implemented via either the JFIF or Exif standards (or an amalgamation of the two)[8]. Most applications are lenient with the file formats they accept and messy with the files they write (see GIMP discussions on Exif[4]).

Instead, we will choose a middle route, opting to use “magic numbers” as our guide. This term refers to tell-tale byte values at predictable offsets within the file data associated with certain file types. 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). See the wikipedia article[9] on magic numbers for more examples.

Writing these definitions in Coq can be relatively straight forward. We need only compare the bytes found within a file (given the context of a specific disk image) to these magic numbers. We will use Coq's `ascii` type to represent a byte; its literal syntax is either a single ASCII character of a three-digit unsigned value. Formalizing the three signatures above and using some Python-like syntactic sugar, this can look like

```
Definition isJpeg (file: File) (disk: Disk) :=
     file @[  0 | disk ] = Found "255"
  ∧  file @[  1 | disk ] = Found "216"
  ∧  file @[- 2 | disk ] = Found "255"
  ∧  file @[- 1 | disk ] = Found "217".

Definition isGzip (file: File) (disk: Disk) :=
     file @[ 0 | disk ] = Found "031"
  ∧  file @[ 1 | disk ] = Found "139"
  ∧  file @[ 2 | disk ] = Found "008".

Definition isElf (file: File) (disk: Disk) :=
     file @[ 0 | disk ] = Found "127"
  ∧  file @[ 1 | disk ] = Found "E"
  ∧  file @[ 2 | disk ] = Found "L"
  ∧  file @[ 3 | disk ] = Found "F".
```

For each definition, the first (and, in the case of the JPEG, last) bytes of a file must both be present (as indicated by `Found`) 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 definitions as building blocks within larger terms 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) (disk: Disk),
  (isJpeg file disk) → ¬(isGzip file disk).
```

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

3. A Definition of Rootkit Installation

Let us 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". We formalize this definition by saying that a rootkit installation must satisfy the following definition:

```
Definition borland_rootkit (disk: Disk) :=
  exists file: File,
    isOnDisk file disk
    ∧  isDeleted file
    ∧  isGzip file disk
    ∧  Tar.looksLikeRootkit (gunzip file) disk.
```

This states that, for the definition to be satisfied, there must exist a deleted, gzipped file on the Honeynet disk image such that, when we unzip that file, the contained tar looks like a rootkit. If such a file exists, the Borland definition is satisfied, and we presume a rootkit has been installed. For the remainder of this section, we dive deeper into each of the definitions in the "and" clause (save `isGzip`, which we have previously described). Section 8 will investigate whether or not Borland's definition is sufficient to describe a rootkit installation.

3.1 The "File" Structure

Each file system, such as Ext2, FAT32, and NTFS, represent "files" differently, yet there are several universal (and somewhat-universal) properties shared across these systems. We use these shared properties to describe a generic, abstract file structure which can be used in proofs without worrying about the underlying file system. This abstraction is common in forensics tools (e.g. the Digital Forensics Framework[3] and the Sleuth Kit[12]) because many properties of files (such as file type, as described above) should not depend on file system traits.

What might these "universal" attributes of a file be? File size (in bytes) seems appropriate, as does deletion status (on most file systems, "deleted" files have a tendency to remain present on disk). We also need a file identifier (which will be file system dependent) to distinguish this file from others on disk. Finally, we have four common-but-not-universal fields - access, modification, creation, and deletion times. These Unix-style timestamps are represented as natural numbers (`N`), and are all optional (indicated by `Exc`,) as not all file systems record their values. The optional fields are particularly useful for constructing event time lines, as described in Section 7.

```
Structure File := mkFile {
  fileId: FileId;
  fileSize: N;
  deleted: bool;
  lastAccess: Exc N;
  lastModification: Exc N;
  lastCreated: Exc N;
  lastDeleted: Exc N
}.
```

The `FileId` type may take multiple forms; for the purposes of our exploration, we care only about Ext2 file systems, files embedded in a tar-archive, and files which are not tied to the disk image (see "assumptions" as defined in Section 4). The first needs an Inode Index to serve as an identifier; the second needs the parent archive's file identifier as well as an offset within the archive; the final will make use of a (potentially partial) list of file data, `list (N*Byte)`, to identify itself.

```
Inductive FileId: Type :=
  | Ext2Id: N →  FileId
  | MockId: list (N*Byte) →  FileId
.
```

What about the file contents? The syntax we used before for accessing file data (`file @[ offset | disk ]`) gets expanded into a file-system-aware function, `fetchByte`. This function uses the provided disk and file identifier to retrieve a requested byte, decoupling the file data from the `File` structure. For Ext2, this means delegating to the appropriate Ext2 function (described in Section 5); for the so-called "Mock" file system, it means searching through the contained list of offsets.

```
Definition fetchByte (fileId: FileId) (disk: Disk)
  : N→ @Fetch Byte :=
  match fileId with
  | Ext2Id inodeIndex ⇒  Ext2.fileByte disk inodeIndex
  | MockId data ⇒  findInPairList data
  end.
```

In an earlier prototype, `Files` carried around a closure containing the relevant file system data. While this was more elegant, printing and constructing `Files` proved to be too unmanageable. We have instead settled on a simple structure which is not directly tied to a disk image. To make the association between images and files, we must require the `isOnDisk` property.

3.2 isOnDisk

How might we define that a file "exists" on a particular disk image? As a first attempt, 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 offset such that every byte afterwards matches that of the file.

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

This definition is not very useful in practice, however, as files are very often fragmented across multiple, disjoint segments of a disk. 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 (logical) byte sequence. Files are stored on disks via file systems, which make no sequential guarantees. Indeed, 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 entirely dependent on the file system in question, however, so we will perform case analysis. While this research focuses on Ext2, we provide the skeleton for other file systems below.

```
Definition isOnDisk (file: File) (disk: Disk):
  match file.(fileId) with
  | Ext2Id inodeIndex ⇒
    (Ext2.findAndParseFile disk inodeIndex) = Found file
  | Fat32Id clusterNumber ⇒
    (Fat32.findAndParseFile disk clusterNumber)
      = Found file
  | BtrfsId key ⇒
    (Btrfs.findAndParseFile disk key) = Found file
  | ...
  | _ ⇒  False
  end.
```

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`). `Found` implies that there were no errors while trying to retrieve the file; see Section 4.2's discussion of the `Fetch` construction for more.

3.3 isDeleted

As our file structure includes a `deleted` boolean value, it should be easy to see that the definition of `isDeleted` need only delegate to that field:

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

3.4 Gunzip, an External Function

Before we can describe `Tar.looksLikeRootkit`, we should note the appearance of an as-yet-undefined `gunzip` function. This function represents the unzipping/deflating operation for gzip-files. As implementing the decompression algorithm within Coq would not be particularly useful to our study, we instead leave its definition open using Coq's `Variable` command. The definition will not be defined anywhere within our programs/evidence and will appear as a hypothesis in the outer-most level of the proof. We will leave confirmation to an external, non-Coq verifier, as discussed in Section 4.4.

3.5 Tars That Look Like Rootkits

The final clause in our definition requires that the provided, unzipped file "looks like a rootkit". Borland described the contents of the archive as containing "necessary" files for a rootkit installation; he went on to provide a list of the file names contained in the archive as evidence. We will intuit that this list was relevant because the file names within the archive looked to match those of system files. To reduce our false-positive rate, we will require that at least two of the file names contained within the tar appear to be system files.

```
Definition looksLikeRootkit (file: File) (disk: Disk) :=
  let fileNames := Tar.parseFileNames file disk in
  exists (filename1 filename2: string),
       filename1 <> filename2
    ∧  In filename1 fileNames
    ∧  In filename2 fileNames
    ∧  (FileNames.systemFile filename1)
    ∧  (FileNames.systemFile 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. This list is clearly not exhaustive; it serves as a simple prototype for future expansion.

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

Use of this definition is straight forward:

```
Lemma systemFile_ex1 :
  systemFile "last/top".

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

3.6 Summary

We have now described `isOnDisk`, `isDeleted`, `isGzip`, and `Tar.looksLikeRootkit` enough that we could see why Borland proposed them as definitions of evidence for a rootkit installation. Along the way, we glossed over several critical parsing/computation functions (including retrieval of bytes in the Ext2 file system and parsing file names from Tar files,) which we will cover in sections 5 and 6. Before diving in, however, we should discuss how one would show that the Honeynet disk image satisfies our property.

4. Proving the Property for a Particular Disk

Now that we have a definition for rootkit installation, we can prove that the disk image provided for the Honeynet competition satisfies that definition. We will use Coq's tactic language to build a proof term to serve as evidence.

```
Lemma borland_honeynet_file:
  forall (disk: Disk),
    honeynet_image_partial ⊆ disk →
      gunzipped_partial f⊆ (gunzip file23 disk) →
        borland_rootkit disk.
  Proof.
    intros disk.
    apply borland_rootkit_witness_subset.
    apply borland_reflection
      with (filename1 := maliciousFileName1)
           (filename2 := maliciousFileName2).
    vm_compute. reflexivity.
Qed.
```

This relatively simple proof hides a great deal of complexity. Why are there hypotheses about `gunzip` and subsets? What are `honeynet_image_partial` and `gunzipped_partial`? How do `borland_rootkit_witness_subset` and `borland_reflection` work? Where did `file23` and the `maliciousFileNames` come from? While we will answer each of these questions in detail, it is important to recognize that writing the individual proof is very straight forward as the lemmas applied would not vary from disk to disk. Indeed, they should be easy to automate, playing a heavy role in the future of this type of research, as discussed in section 10. Further, the proof term itself is quite small, the importance of which will be covered in section 4.5.

4.1 Disk Images

While our proof will ultimately show validity over the whole disk image, we will only compute over a *partial* segment, dubbed `honeynet_image_partial`. Both it and the larger image have the type, `Disk`, which is used to represent random-access throughout our definitions and proofs. Technically, `Disk`s are functions from `N` to `Fetch Byte`, acting as a byte-retrieval mechanism; give a `Disk` an offset, and it will respond with the byte value at the offset requested or an error code. The `ByteData` type is simply an alias of `Disk`, but provides a more semantic name.

Generally, disk data is backed by an in-memory mapping via Coq's `FMapAVL` tree, 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":Byte),
  1025 |→ ("002":Byte),
  1026 |→ ("001":Byte),
  1027 |→ ("000":Byte),
  ...
].

Definition Disk_of_Map_N_Byte (map: Map_N_Byte) : Disk :=
  fun (key: Z) ⇒ find key map.

Definition honeynet_image_partial : Disk :=
  Disk_of_Map_N_Byte honeynet_map.
```

Using this method, we need only the sections of disk crucial to the file system's structure, resulting in an in-memory mapping which is a fraction of the size of the original disk (hence the suffix, `_partial`).

Disk Size (Bytes)	
Size of Disk Image	271,401,984
Size of Relevant Map	2,404

4.2 The Fetch Type and Populating Disk Images

As disk images are too large to place in memory, the `Disk` discussed above will have "gaps." We initially gravitated towards the Option monad (`Exc` in Coq,) but realized that, if we were to chain options (see 5.1,) we would have no feedback on *where* the retrieval operations failed. We therefore created a new inductive type, `Fetch`, which can be instantiated as a `Found` when the byte (or other type) is present, a `MissingAt` when not present, and an `ErrorString` when a different error occurs.

```
Inductive Fetch {A:Type}: Type :=
  | Found: A → Fetch
  | MissingAt: N → Fetch
  | ErrorString: string → Fetch
.
```

To see why carrying along *where* a fetching operation failed is useful, let us consider how a disk image is constructed. The Coq language does not provide a mechanism for file access, so we cannot write a program within it to populate our Disk image. The definitions and functions that define what data is needed are only available within Coq, however, so while we could use Python or Scala or any general-purpose language to retrieve necessary bytes, those languages would not know which bytes were actually necessary.

Luckily, Coq has an `Extraction` command, which allows a Coq function/data structure to be exported to Ocaml, Haskell, or Scheme. The extraction process can include all necessary types and referenced terms to build the requested expression. With this ability, we can use the definitions created in Coq in concert with file access built into those languages to construct an entire disk structure. We would simply extract a function which requires the disk image, feed it an empty `Disk`, see where (`MissingAt`) the function failed, retrieve that byte from disk, add it to the `Disk` structure, and repeat until the function was successful.

4.3 Subsets and Monotonicity

While proving the lemma holds for a given partial disk image is valuable, our argument becomes stronger if we can prove the lemma for a *whole* disk. As we cannot load the entirety of the disk into Coq, we will proceed instead via a "monotonicity" proof. By this we mean that we will show that if particular lemmas apply for partial images, they also apply for "whole" disk image. Using these monotonicity proofs (such as `borland_rootkit_witness_subset`), we need only process the partial disk image.

Let us first consider what it would mean for a `Disk` to be a subset of another. This is relatively straight forward; if we were to find a byte on the partial, we would expect the same value to be present on the larger disk.

```
Definition disk_subset (sub super: Disk) :=
  forall (offset: N) (byte: Byte),
    sub offset = Found byte → super offset = Found byte.

Infix "⊆" := disk_subset (at level 50).
```

Subsets in the context of `Files` are a bit more complicated, as file access depends on the file system. If using the "Mock" file system, we will proceed similar to `disk_subset`, verifying that when the byte is present in the partial, the value is the same in the larger `File`. Otherwise, the definition need only compare the fields of the `File` for equality.

```
Definition fileId_subset (sub super: FileId) :=
  match (sub, super) with
    | (Ext2Id _, Ext2Id _) ⇒ sub = super
    | (MockId subData, MockId superData) ⇒
        forall (idx: N) (byte: Byte),
          (findInPairList subData idx) = Found byte →
            (findInPairList superData idx) = Found byte
    | _ ⇒ False
end.

Definition file_subset (sub super: File) :=
  fileId_subset sub.(fileId) super.(fileId)
  ∧ sub.(fileSize) = super.(fileSize)
  ∧ sub.(deleted) = super.(deleted)
  ∧ sub.(lastAccess) = super.(lastAccess)
  ∧ sub.(lastModification) = super.(lastModification)
  ∧ sub.(lastCreated) = super.(lastCreated)
  ∧ sub.(lastDeleted) = super.(lastDeleted).

Infix "f⊆" := file_subset (at level 50).
```

Monotonicity will not apply for every property; imagine a property that checked if a byte was present, for example. This means that we will need to explicitly provide monotonicity proofs (as indicated by the "`_subset`" suffix) on demand (and for many properties). We will leave a full accounting of these properties and proofs to the provided source code, but provide two examples here.

```
Lemma isOnDisk_subset :
  forall (sub super: Disk) (file: File),
    sub ⊆ super →
```

```
      isOnDisk file sub →
        isOnDisk file super.

Lemma looksLikeRootkit_fsubset:
  forall (sub super: File) (disk: Disk),
    sub f⊆ super →
      looksLikeRootkit sub disk →
        looksLikeRootkit super disk.
```

4.4 The Gunzip Hypothesis

As we just saw, Coq's proof-oriented nature has left the language with 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 in memory. We work around these limitations in part by using only subsets of the disk image and file structures which entail only the relevant pieces of data needed for our proofs. As discussed above, we will not read a full disk image; instead, we will generate a *sparse* map to represent that disk, including only the offsets of interest.

We also use hypotheses to account for a transformations that would normally be performed by a trusted, external program. We cannot run "gunzip", nor do we want to try to implement that algorithm within Coq. Instead, we describe an undefined implementation to encompass that activity. Our proofs cannot validate that the unzipping operation was successful within Coq; their statements must instead carry the caveat that the data provided in `gunzipped_partial` is accurate. It is up to the users (or programs; see Section 10) to verify these hypotheses.

This said, we want to verify as much as possible in our implementation, so we will provide data from the real, unzipped file via the `gunzipped_partial` structure. This is a constant `File` which has been pre-populated with the relevant bytes using the "Mock" file identifier. We generated the data needed for this file by using the *Sleuth Kit*'s `icat` program to pull inode 23's contents. We then ran `gunzip` on the resulting `tgz` file, and pulled the relevant bytes (i.e. those necessary for the tar headers, see Section 6) into the `File`. The latter step could be automated using `Fetch` as described in the previous section.

File Size (Bytes)	
Inode 23	520,333
Gunzipped Inode 23	1,454,080
`gunzipped_23`	580

```
Definition gunzipped_23 : Map_N_Byte :=
[
  0 |→  ("108":Byte),
  1 |→  ("097":Byte),
  2 |→  ("115":Byte),
  ...
].

Definition gunzipped_partial : File := mkFile
  (FileIds.MockId (MN.elements gunzipped_23))
  1454080 (* uncompressed file size *)
  true
  (* Fields not used; ignore them *)
  None None None None.
```

4.5 Proof By Reflection

Each of the definitions we have seen so far lives within Coq's `Prop` realm. For properties to be proven valid, Coq's proof tactics are required, describing how assumptions can be transformed to imply the conclusion. This lies in contrast to Coq's boolean type, which can be **computed**, meaning boolean expressions can be reduced to a single value. To make the distinction clear, let us consider two analogous goals involving a simple reduction.

```
Lemma reduce_prop:
  forall (P Q R :Prop),
  Q →  R →  ¬P →
    (Q ∨  ¬R) ∧  (P ∨  (¬P ∧  R ∧  Q)).
Proof.
  intros.
  split.
    (* Q ∨  ¬R *)
    left. assumption.
    (* P ∧  (¬P ∧  R ∧  Q) *)
    right.
    split.
      (* ¬ P *) assumption.
    split.
      (* R *) assumption.
      (* Q *) assumption.
Qed.

Lemma reduce_bool:
  forall (p q r :bool),
  q = true  →  r = true  →  p = false  →
    (q || (negb r)) && (p || ((negb p) && r && q)) = true.
Proof.
  intros. rewrite H. rewrite H0. rewrite H1.
  compute.
  reflexivity.
Qed.
```

The latter is largely a matter of plugging in values followed by the `compute` tactic. Since it deals only with boolean logic, which can be simplified via automated reduction techniques[2], we do not need to tease it apart as we do with the `Prop` version. This makes sense, as `Prop` represents more than just boolean values; how could we *compute* that a property held for all integers, or that there *exists* a value that satisfies a certain property?

Not all properties are this complex, however, particularly in our problem domain. We are largely proving that byte values are equivalent, meaning it would be quite helpful to rely on boolean algebra in our proofs. We do not want the definitions to be entirely composed of booleans, however, as boolean phrases have significantly less expressive power. If we were able to prove that a boolean reduction *implied* a property, however, we could then prove that property simply by running `compute`. This is the essence of proof by reflection.

As an example, consider our `isJpeg` property. We have also written an equivalent, boolean version, `isJpeg_compute` and a lemma, `isJpeg_reflection` which shows how the boolean version implies the property. Note that `Byte.feqb` is a boolean form of equality over `Fetch Bytes`.

```
Definition isJpeg (file: File) (disk: Disk) :=
     file @[  0 | disk ] = Found "255"
  ∧  file @[  1 | disk ] = Found "216"
  ∧  file @[- 2 | disk ] = Found "255"
  ∧  file @[- 1 | disk ] = Found "217".

Definition isJpeg_compute (file: File) (disk: Disk) :=
     Byte.feqb (file @[  0 | disk ]) (Found "255")
  && Byte.feqb (file @[  1 | disk ]) (Found "216")
  && Byte.feqb (file @[- 2 | disk ]) (Found "255")
  && Byte.feqb (file @[- 1 | disk ]) (Found "217").

Lemma isJpeg_reflection (file: File) (disk: Disk) :
  isJpeg_compute file disk = true →  isJpeg file disk.
```

Using proof by reflection (as we do in `borland_reflection`) not only reduces the number of proof tactics necessary, it also reduces the size of the so-called "proof term." The proof term is the ultimate evidence that a property holds; it is what can be shipped around and verified in our hypothetical court cases. If the size of this proof term grew proportional to the size of the disk image, the terms would quickly become unmanageable. Plausible use of these techniques would likely mean applying properties over thousand (if not millions) of files; only if the proof terms remain a constant size can they be useful at this scale. Proof by reflection moves us further towards accomplishing that goal.

4.6 Computed Structures

While the definitions we use are conducive to conversion into boolean forms, there are a few situations where we would still like the `Prop` version to include an existential variable. Our definition of a tar file which `looksLikeRootkit`, for example, states that there *exists* two distinct filenames that satisfy certain properties.

```
Definition looksLikeRootkit (file: File) (disk: Disk) :=
  let fileNames := parseFileNames file disk in
  exists (filename1 filename2: string),
    filename1 <> filename2
    ∧  In filename1 fileNames
    ∧  In filename2 fileNames
    ∧  (FileNames.systemFile filename1)
    ∧  (FileNames.systemFile filename2).
```

Using existential variables here makes the definition cleaner. It does not really matter what the file names are; the definition only requires *that* they are. How can this be converted into a computational version? If we were proving that certain file names satisfied such a definition, we would generally provide the names as existential proof (via the `exists` tactic). We will need to do the same for the boolean function, providing the file names as parameters.

```
Definition looksLikeRootkit_compute (file: File) (disk: Disk)
  (filename1 filename2: string) :=
  let fileNames := parseFileNames file disk in
     (negb (string_eqb filename1 filename2))
  && (existsb (string_eqb filename1) fileNames)
  && (existsb (string_eqb filename2) fileNames)
  && (FileNames.systemFile_compute filename1)
  && (FileNames.systemFile_compute filename2).
```

We also need a lemma that ties the compute version to the `Prop` version; this lemma will likewise need the file names.

```
Lemma looksLikeRootkit_reflection (file: File) (disk: Disk)
  (filename1 filename2: string) :
  looksLikeRootkit_compute file disk filename1 filename2 = true
    →  looksLikeRootkit file disk.
```

When we apply `looksLikeRootkit_reflection`, we will provide the two file names as parameters (instead of providing them to the `exists` tactic). Where do these existential values come from? For our examples, we simply instantiated the file name strings and file structures directly, but the longer-term strategy is to compute them from the assumptions. These computed structures are *not* assumptions, but could easily be created during the assumption generation phase. See Section 10 on future work for details.

5. Ext2 Structures and Computations

In proving that a file was present on an Ext2 disk image, we deferred to the simple interface of `Ext2.findAndParseFile`; similarly, when determining individual bytes of a file, we relied on the `Ext2.fileByte` function. Unfortunately, these interfaces hide a great deal of complexity present within Ext2 disks. As a proof of concept, we next provide implementation details for these functions; it should be easy to see how this research could be extended to include additional file systems so long as similar interfaces can be provided.

5.1 A Note on Functional Idioms

While computing, 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. We use the `Fetch` type (discussed in Section 4.2) to represent these options.

The benefit of wrapping the value in the monadic `Fetch` is that we can continue computation without knowing (in advance) whether a sub-computation was successful or not. `Fetch`s make error handling "percolate up" in that functions which produce them can be chained such that *any* error in a component causes the entire computation to result in that error. To get to this point, we define two functions, indicated by the infix notation `_fmap_` and `_fflatmap_` (the prepending 'f' indicates that the `Fetch` type is used rather than Coq's built-in `Exc`). The function signatures for each should aid their explanation.

```
Definition fetch_map {A B: Type} (opt: @Fetch A) (fn: A →  B)
  : Fetch B.

Definition fetch_flatmap {A B: Type} (opt: @Fetch A)
  (fn: A →  @Fetch B)
  : @Fetch B.
```

The role of `_fmap_` is to transform the contents of a `Fetch`, if present. If the `Fetch` contains an error, `_fmap_` has no effect. `_fflatmap_` similarly does not affect such `Fetch`s. However, if a value is present (indicated by `Found`), the provided function (`fn`) is applied and the `Fetch` is replaced with that function's result. In other words, `_fflatmap_` is like applying an `_fmap_` and then removing the outer `Found`. These functions allow sequences like:

```
(file @[ 0 | disk ]) _fflatmap_ (fun byte0 ⇒
(file @[ 1 | disk ]) _fflatmap_ (fun byte1 ⇒
(file @[ 2 | disk ]) _fmap_ (fun byte2 ⇒
  (byte0, byte1, byte2)
)))
```

which can return an `ErrorString`; a `MissingAt` with 0, 1, or 2; or `Found (Byte, Byte, Byte)` (all three bytes present). If the first byte was not present, the outer-most `_fflatmap_` would not have executed the inner function. Similarly, if the second byte were not present, the function including `_fmap_` would not be ran, and if the third byte were unavailable, the function with the parameter `byte2` would not be evaluated.

We have only barely touched 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[13] or Minsky, Madhavapeddy, & Hickey[10].

5.2 findAndParseFile

To explain our two functions, we will need to understand how Ext2 file systems are laid out on disk. We will proceed by removing each layer of the calls and review the data structures created.

At the outermost layer of `findAndParseFile`, we have a function which, when given a disk and Ext2 Inode identifier, returns a `Fetch File`. As we discussed earlier, a `File` is composed of a file-system-specific identifier, file size, deletion status, and a few time-related values. These fields require we first parse out the `SuperBlock` of the disk as well as the `GroupDescriptor` and `Inode` associated with our Inode index (the unique file identifier for the Ext2 file system).

To understand what these structures represent, let us discuss Ext2, as described by one of its original authors, David Poirier[11]. 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). Here, we should note that for both of these operations, 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 retrieve 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 `_fflatmap_` approach described above, treating the computation as a monad.

```
Definition findAndParseFile (disk: Disk) (inodeIndex: N)
  : @Fetch File :=
  (findAndParseSuperBlock disk) _fflatmap_ (fun superblock ⇒
  let groupId := ((inodeIndex - 1) (* One-indexed *)
                 / superblock.(inodesPerGroup)) in
  let inodeIndexInGroup :=
    (inodeIndex - 1) mod superblock.(inodesPerGroup) in
  (findAndParseGroupDescriptor disk superblock groupId)
    _fflatmap_ (fun groupdesc ⇒
  (findAndParseInode disk superblock groupdesc inodeIndex)
    __flatmap_ (fun inode ⇒
  (parseDeleted disk superblock groupdesc inodeIndex)
    _fmap_ (fun deleted ⇒
    mkFile
      (FileIds.Ext2Id (value inodeIndex))
      inode.(size)
      deleted
      (value inode.(atime))
      (value inode.(mtime))
      (value inode.(ctime))
      (value inode.(dtime))
  )))).
```

Discarding the outer layer, we next take a look at each of the composing computations, `findAndParseSuperBlock`, `findAndParseGroupDescriptor`, `findAndParseInode`, and `parseDeleted`.

5.3 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 format is largely compatible, so we will use the SuperBlock structure described by Poirier. This specification cares only about the first 264 or so bytes of the block (regardless of its full size).

Finding and parsing the SuperBlock, then, amounts to jumping to offset 1024 on the disk and placing the read values into a SuperBlock structure. To make this operation smoother, 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: N)
  : @Fetch Byte :=
  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 `N` 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)
  : @Fetch SuperBlock :=
  let disk := (shift disk 1024) in
  (seq_lendu disk 0 4) _fflatmap_ (fun inodesCount ⇒
  (seq_lendu disk 4 4) _fflatmap_ (fun blocksCount ⇒
  (* ... Additional fields omitted ... *)
  (seq_lendu disk 260 4) _fmap_ (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) :=
  N.shiftl 1024 superblock.(logBlockSize).
```

5.4 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 := N.

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

5.5 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: N)
  : @Fetch GroupDescriptor :=
  let groupBlockArrayBA := if (1024 <? blockSize superblock)
    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) _fflatmap_ (fun blockBitmap ⇒
  (seq_lendu disk 4 4) _fflatmap_ (fun inodeBitmap ⇒
  (seq_lendu disk 8 4) _fflatmap_ (fun inodeTable ⇒
  (seq_lendu disk 12 2) _fflatmap_ (fun gdFreeBlocksCount ⇒
  (seq_lendu disk 14 2) _fflatmap_ (fun gdFreeInodesCount ⇒
  (seq_lendu disk 16 2) _fmap_ (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.

5.6 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 pinpoint 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: N) : @Fetch Inode :=
  (* Check for valid Inode *)
  if (superblock.(inodesCount) <=? inodeIndex)
  then ErrorString "Invalid inode index"
  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) _fflatmap_ (fun mode ⇒
    (* ... Additional fields omitted ... *)
    (seq_lendu disk 40 4) _fflatmap_ (fun directBlock1 ⇒
    (* ... Additional direct blocks omitted *)
    (seq_lendu disk 84 4) _fflatmap_ (fun directBlock12 ⇒
    (seq_lendu disk 88 4) _fflatmap_ (fun indirectBlock ⇒
    (seq_lendu disk 92 4) _fflatmap_ (fun
                                  doubleIndirectBlock ⇒
    (seq_lendu disk 96 4) _fflatmap_ (fun
                                  tripleIndirectBlock ⇒
    (* ... Additional fields omitted ... *)
    (seq_lendu disk 116 4) _fmap_ (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.

5.7 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 containing byte within that bitmap, read it, and test the relevant bit within 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: N) : @Fetch 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)))
    _fmap_ (fun allocationByte ⇒
    (* The bit associated with this inode is 0 *)
    match (allocationByte, inodeIndexInGroup mod 8) with
    | (Ascii b _ _ _ _ _ _ _, 0) ⇒  negb b
    | (Ascii _ b _ _ _ _ _ _, 1) ⇒  negb b
    | (Ascii _ _ b _ _ _ _ _, 2) ⇒  negb b
    | (Ascii _ _ _ b _ _ _ _, 3) ⇒  negb b
    | (Ascii _ _ _ _ b _ _ _, 4) ⇒  negb b
    | (Ascii _ _ _ _ _ b _ _, 5) ⇒  negb b
    | (Ascii _ _ _ _ _ _ b _, 6) ⇒  negb b
    | (Ascii _ _ _ _ _ _ _ b, 7) ⇒  negb b
    | _ ⇒  false  (* should never be reached *)
    end
  ).
```

With these functions, we can create a `File` structure from an Ext2 file system. Next we discuss how to retrieve an arbitrary byte within the file given that structure.

5.8 fetchByte Refresher and fileByte

Recalling, `fetchByte` is our interface for retrieving a particular byte from a file structure. This interface treats files as linear sequences of bytes, regardless of their layout on disk. It effectively delegates to the `Ext2.fileByte` function.

```
Notation "f @[ i | d ]" :=
  (fetchByte f.(fileId) d i) (at level 60).

Fixpoint fetchByte (fileId: FileId) (disk: Disk)
  : N→ @Fetch Byte :=
  match fileId with
  | Ext2Id inodeIndex ⇒  Ext2.fileByte disk inodeIndex
  | ...
  end.
```

This function needs the same type of information needed to construct the `File` structure. Unfortunately, we will recreate those same structures *every* time we retrieve a byte. This certainly will slow down our computations, but provides a very clean interface for accessing file data. We split the function into an outer layer, very similar to `findAndParseFile` and a deeper function, `fetchInodeByte`, which handles the new concepts.

```
Definition fileByte (disk: Disk) (inodeIndex offset: N)
  : @Fetch Byte :=
  (findAndParseSuperBlock disk) _fflatmap_ (fun superblock ⇒
  let groupId := ((inodeIndex - 1) (* One-indexed *)
                  / superblock.(inodesPerGroup)) in
  let inodeIndexInGroup :=
    (inodeIndex - 1) mod superblock.(inodesPerGroup) in
  (findAndParseGroupDescriptor disk superblock groupId)
    _fflatmap_ (fun groupdesc ⇒
  (findAndParseInode disk superblock groupdesc inodeIndex)
    _fflatmap_ (fun inode ⇒
      fetchInodeByte disk superblock inode offset
  ))).
```

5.9 fetchInodeByte

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 retrieve 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: N) (indirectionLevel: nat)
  : @Fetch BA :=
  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.

```
Fixpoint walkIndirection (disk: Disk) (superblock: SuperBlock)
  ...
  | S nextIndirectionLevel ⇒
    (* Type conversion *)
    let exponent := N.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 multiple 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: N): @Fetch Byte :=
  if inode.(size) <=? bytePos then
    MissingAt bytePos
  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 various 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)
    ...
    (if blockNumInFile <=? directAddressable then
      match (nth_error inode.(block)
                       (N.to_nat blockNumInFile)) with
      | error ⇒  ErrorString "Data block not present"
      | value v ⇒  Found v
      end
    ...
```

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
      match (nth_error inode.(block) 12) with
      | error ⇒  ErrorString "Indirection block not present"
      | value indirectBlock ⇒
        walkIndirection disk superblock
          (blockNumInFile - 12)
          (ba2Offset superblock indirectBlock)
          O
      end
    ...
```

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
      match (nth_error inode.(block) 13) with
      | error ⇒ ErrorString
                  "Double indirection block not present"
      | value doubleIndirectBlock ⇒
        walkIndirection disk superblock
          (blockNumInFile - 12 - (blockSize / 4))
          (ba2Offset superblock doubleIndirectBlock)
          (S O)
      end

    else
      match (nth_error inode.(block) 14) with
      | error ⇒ ErrorString
                  "Triple indirection block not present"
      | value tripleIndirectBlock ⇒
        walkIndirection disk superblock
          (blockNumInFile - directAddressable
                          - indirect1Addressable
                          - indirect2Addressable)
          (ba2Offset superblock tripleIndirectBlock)
          (S (S O))
      end
    )
    ...
```

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)
    ...
    (
      ...
    ) _fflatmap_ (fun blockAddress ⇒
      disk (blockSize * blockAddress + (bytePos mod blockSize))
    ).
```

With this function, we can conclude our discussion of the components needed to both construct `File` structures and retrieve bytes using them from an Ext2 disk. We will next describe similar computation methods for tar archives.

6. Tar Archive Structures and Computations

Our definition of rootkit installation made use of `Tar.parseFileNames`, a function which retrieves a list of all file names includes from a tar archive. As with the Ext2 file system, retrieving these file names requires a discussion of the tar file format. Unlike our Ext2 studies, which described the layout of a file system on a disk image, tar files are constructed within the context of a particular `File`. This means that we will only access bytes on the disk through the `file @[ offset | disk ]` construct (or through `fetchByte` directly); 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 one file name from the tar header, read the file size from that header, skip the file contents and start again with the following header.

6.1 Parsing a Single Header

First, we will write a function that can pull out a filename and associated file size. As fetching either could fail, we will chain the two actions via a flatmap. We will provide this function a tar file as well as an offset within that tar file to indicate where the header in question begins. Fetching all pairs will be a matter of shifting the "start" of the file again and again.

```
Definition parseFileNameAndSize (tar: File) (offset: N)
  (disk: Disk): @Fetch (string*N) :=
  (parseFileName tar offset disk) _fflatmap_ (fun name ⇒
  (parseFileSize tar offset disk) _fmap_ (fun size ⇒
    (name, size)
  )).
```

6.2 File Name

Retrieving the file name requires reading the first hundred bytes of the header and taking every character until the string is terminated (via a null). As each attempt to read from the file could fail, we will read each byte in sequence and only continue if a byte was read successfully. We then recurse until the null byte is hit, we have reached our hundred-byte limit, or an error occurs. To implement this, we will used `N.peano_rect`, the details of which are not relevant to this study. Instead, we will highlight the recursive call embedded within it.

```
Definition parseFileName (tar: File) (offset: N)
  (disk: Disk): @Fetch string :=
  ...
  match (tar @[ offset + idx | disk ]) with
  | Found byte ⇒
    if (ascii_eqb byte Ascii.zero) (* Null *)
    then Found EmptyString
    else match (rec (idx + 1)) with
    | Found rest ⇒ Found (String byte rest)
    | MissingAt val ⇒ MissingAt val
    | ErrorString val ⇒ ErrorString val
    end
  | MissingAt val ⇒ MissingAt val
  | ErrorString val ⇒ ErrorString val
  end
  ...
```

6.3 File Size from ASCII Octal

The size of a file within a tar can be determined by reading the 11 bytes starting at offset 124 of the associated 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 most likely relates to standards at the time of the file format's construction. We implement our parsing with a simple recursive function which reads the next byte (i.e. character) from a list and converts that into the encoded integer value. The value is multiplied by 8 raised to the length of the remaining 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. Note that `value` and `error` are the two subtypes of `Exc`, Coq's native option type.

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

We will attempt to read each of the eleven bytes of the file size header, and, if all goes well, pass that on to the `fromOctalAscii` function described above. We must also account for a special case: tar files are sometimes padded to additional multiples of 512 via null bytes. This means that we will need to account for a file "size" indicated by all null bytes and equate that to a zero-length entry.

```
Definition parseFileSize (tar: File) (offset: N) (disk: Disk)
  : @Fetch N :=
  (tar @[ offset + 124 | disk ]) _fflatmap_ (fun byte0 ⇒
  (tar @[ offset + 125 | disk ]) _fflatmap_ (fun byte1 ⇒
  ...
  (tar @[ offset + 134 | disk ]) _fflatmap_ (fun byte10 ⇒
    match (fromOctalAscii
            (byte0 :: byte1 :: byte2 :: byte3 :: byte4
             :: byte5 :: byte6 :: byte7 :: byte8 :: byte9
             :: byte10 :: nil)) with
    | error ⇒
        (* the last few chunks of the Tar file may be full
           of zeros *)
        if ((Byte.eqb byte0 Ascii.zero)
             && (Byte.eqb byte1 Ascii.zero)
             ...
             && (Byte.eqb byte10 Ascii.zero))
          then Found 0
          (* something went wrong *)
          else ErrorString "Invalid Tar Size"
    | value size ⇒  Found size
    end))))))))))).
```

6.4 Recursing Through

Now that we have a function which fetches one filename and size from the tar, we can call it repeatedly, moving the initial offset with each step. We need to ignore the previous file header (512 bytes) plus the file size padded to 512 bytes. We will not 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. To see the full definition of `Tar.parseFileNames`, please see the source code.

```
Definition recFileNameFrom (nextCall: N→ @Fetch (list string))
  (tar: File) (offset: N) (disk: Disk) : @Fetch (list string) :=
  if (tar.(fileSize) <=? offset)
    then Found nil
  else match
    (parseFileNameAndSize tar offset disk) with
    | Found (fileName, fileSize) ⇒
        (* Strip the next file out of the tar *)
        (* Round to the nearest 512 *)
        let nextFileSize := (
          if (fileSize mod 512 =? 0)
          then fileSize + 512
          else 512 * (2 + (fileSize / 512))) in
        match (nextCall (offset + nextFileSize)) with
        | Found rest ⇒  Found (fileName :: rest)
        | MissingAt val ⇒  MissingAt val
        | ErrorString val ⇒  ErrorString val
        end
    | MissingAt val ⇒  MissingAt val
    | ErrorString val ⇒  ErrorString val
    end.
```

With this (and `N.peano_rect`) we can construct `parseFileNames` to retrieve a list of the file names (as strings) contained within a tar file. In an effort to check the efficiency of this function, we now provide the running times for its application at various sizes of Tar headers. We will construct partial tar files with different header counts and test how long a Linux netbook takes to compute their file names. For details on how the tests were performed, see the `tar_runtimes.v` file in the associated repository.

parseFileNames Running Times	
Number of Headers	Running Time (seconds)
1	0.0282
5	0.0353
10	0.0805
50	1.0157
100	3.5628
500	91.3177
1000	332.9175

Clearly this is not as fast (and scales more poorly) than the `tar` utility, but this performance should also be acceptable for most archive files.

6.5 Computing Over File Names

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 use 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: string): string :=
  let reversedName := rev fileName in
  list2string
    (rev (takeWhile (fun (char: ascii) ⇒
                       (negb (orb (ascii_eqb char "/")
                                  (ascii_eqb char "\"))))
                    reversedName)).
```

7. Timelines as Evidence

A second type of evidence provided by the Honeynet researchers (particularly Jason Lee[7]) has 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.

```
Definition Timeline: Type := list Event.
```

7.1 Events and Their Relations

The `Event` type currently consists of four forms, representing four file-related 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: N → FileId → Event
  | FileModification: N → FileId → Event
  | FileCreation: N → FileId → Event
  | FileDeletion: N → FileId → Event
```

While not all conceivable events have a timestamp, those that do make the `beforeOrConcurrent` definition significantly simpler. We provide a helper function `timestampOf`, which retrieves the timestamp for all existing `Event` types. With that, proving one event happend `beforeOrConcurrent` with another is a simple matter of comparing timestamps.

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

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. In this situation, an `Event` might include a line number, which could be compared to others.

7.2 The Existence of Events

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) :=
  match event with
  | FileAccess timestamp fs ⇒  exists (file: File),
        isOnDisk file disk
        ∧  fs = file.(fileId)
        ∧  file.(lastAccess) = value timestamp
  | FileModification timestamp fs ⇒  exists (file: File),
        isOnDisk file disk
        ∧  fs = file.(fileId)
        ∧  file.(lastModification) = value timestamp
  | FileCreation timestamp fs ⇒  exists (file: File),
        isOnDisk file disk
        ∧  fs = file.(fileId)
        ∧  file.(lastCreated) = value timestamp
  | FileDeletion timestamp fs ⇒  exists (file: File),
        isOnDisk file disk
        ∧  fs = file.(fileId)
        ∧  file.(lastDeleted) = value timestamp
  end.
```

7.3 Soundness

To verify that a Timeline is sound, we will defer to each consecutive pair of events. If each of these pairs is sound, we declare the entire Timeline to be as well. The `combine` operator "zips" two lists together, creating a list of pairs; the combined list is the length of the shorter input. We use that along with `skipn` (which drops the requested number of elements from the head of a list) to pair each event such that the first event is paired with the second, the second with the third, and so forth.

```
Definition isSound (timeline: Timeline) (disk: Disk) :=
  let staggeredEvents := combine timeline (skipn 1 timeline) in
  forall (pair: Event*Event),
    In pair staggeredEvents →  isSoundPair disk pair.
```

The final piece to defining a sound Timeline is to define `isSoundPair`, which simply needs to show that both events are found on disk and that the first is `beforeOrConcurrent` with the second.

```
Definition isSoundPair (disk: Disk) (eventPair: Event*Event) :=
  let (lhsEvent, rhsEvent) := eventPair in
  foundOn lhsEvent disk
  ∧  foundOn rhsEvent disk
  ∧  beforeOrConcurrent lhsEvent rhsEvent = true.
```

7.4 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; in the conversion we lose file name (instead, we use inode number) and pretty-printed dates (opting for Unix timestamps).

```
Definition lee_timeline :=
  (* Mar 16 01 12:36:48 *)
    (* rootkit lk.tar.gz downloaded *)
      (FileModification 984706608 (Ext2Id 23))
  (* Mar 16 01 12:44:50 *)
    (* Gunzip and Untar rootkit lk.tar.gz *)
      :: (FileAccess 984707090 (Ext2Id 23))
    (* change ownership of rootkit files to root.root *)
      :: (FileAccess 984707102 (Ext2Id 30130))
    (* deletion of original /bin/netstat *)
      :: (FileDeletion 984707102 (Ext2Id 30188))
    (* insertion of trojan netstat *)
      :: (FileCreation 984707102 (Ext2Id 2056))
    (* deletion of original /bin/ps *)
      :: (FileDeletion 984707102 (Ext2Id 30191))
    (* insertion of trojan ps *)
      :: (FileCreation 984707102 (Ext2Id 2055))
    (* deletion of origin /sbin/ifconfig *)
      :: (FileDeletion 984707102 (Ext2Id 48284))
    (* insertion of trojan ifconfig *)
      :: (FileCreation 984707102 (Ext2Id 2057))
  (* Mar 16 01 12:45:03 *)
    (* the copy of service files to /etc *)
      :: (FileAccess 984707103 (Ext2Id 30131))
    (* hackers services file copied on top of original *)
      :: (FileCreation 984707103 (Ext2Id 26121))
  (* Mar 16 01 12:45:05 *)
    (* deletion of rootkit lk.tar.gz *)
      :: (FileDeletion 984707105 (Ext2Id 23))
  :: nil.
```

```
Lemma lee_honeynet_file:
  forall (disk: Disk),
    honeynet_image_partial ⊆ disk →
      Timeline.isSound lee_timeline disk.
```

As with `borland_rootkit`, proving this lemma involves applying a monotonicity proof, a reflection proof, and then computing. See the code linked in the appendix for details.

8. 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. This assumption is not much of a stretch; we would simply need to extend our Ext2 definitions to retrieve file names in addition to file structures.

With this assumption, are there scenarios where our definitions could be satisfied without a rootkit being installed? Let us 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 would not 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 is 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 satisfy 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 tell-tale fingerprints, either within the executable's bytes or within its actions. Unfortunately, as anti-virus software vendors have found, constantly evolving signatures are quite difficult to maintain. This strategy is not sustainable in the long term and certainly falls outside of the scope of this research.

Indeed, any definition for root kit installation which relies solely on the disk image could be satisfied given a random number generator and enough iterations or direct write access. In either case, the *exact* data that would indicate a root kit installation could be present on disk. Clearly, no such software would have been installed, yet our definitions would be satisfied. Ultimately, the definitions we provide cannot guarantee their semantic meaning. It is up to the forensics community to determine how much evidence qualifies an "enough".

This, too, is the methodology found within the Honeynet competition; as there were no canonical definitions for what it means for a root kit 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. Judges of the competition served as the arbiters, deciding which evidence was sufficient for the winning entries.

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.

That said, these definitions are more powerful than those found in the Honeynet researchers' write-ups. Our definitions live in the land of formal, declarative, composable properties. This means they can serve as building blocks for larger properties, combined into general purpose lemmas, and de-constructed into independent components. In this form, forensics researchers can not only make concrete statements about their results and theorize about potential implications but they can also constructively share those atoms, building a wealth of reusable knowledge not present in one-off competitions.

9. Tractability

The final artifact when proving a lemma is a so-called "proof term", a type-checkable entity which serves as verifiable evidence for the lemma. If this proof term were to grow with the size of the underlying disk (or files or file headers, etc.,) it would quickly become unmanageable. Larger disks would eventually prevent a proof-checker from running, making the whole effort moot. Given the orders of magnitude involved (e.g. scanning hundreds of thousands of files, verifying the disk structure of multi-terabyte drives, etc.,) a tool that cannot scale is already useless.

Instead, the design presented (which utilized proof by reflection) causes a well-developed proof to grow only with the size of the *lemmas*. Regardless of disk size, the proof term can remain fixed (though referring to the disk), reducing the burden for proof-checkers. Further, with a consistent proof term, we open the possibility of expanding beyond some of Coq's limitations. As discussed previously, due to Coq's representations, relevant sections on disk must be limited to a few thousand offsets. However, we could hypothetically export a proof term generated within Coq on a smaller disk size and run it through an external proof-checker to handle larger disk images.

Indeed, we can use the external tool `coqchk` (a separate, tighter implementation of Coq's semantics) to verify our work. Our calls to the optimized `vm_compute` function are replaced by a significantly slower computation engine (on the order of 100-fold increase in run time). However, our tests have proven that using an external tool is possible – we do not use any special tactics, etc.

The size of the proof term is one important metric for tractability, but let us not forget execution time. On our relatively small examples (with a disk image containing only a couple of thousand bytes), execution time is reasonable when both proving the honeynet lemmas (via compute) and verifying them (via Qed). Further, adding non-relevant bytes to the disk image appears to have little effect on type-checking (i.e. verifying the proof). While adding extra bytes does not heavily influence the computation steps in the proof of root kit installation, it does appear to have negative consequences to the timeline verification proof. Of course, the more we are trying to prove, the more we will need to compute, and the longer we can expect our computations to run.

Malicious Archive Running Times (seconds)		
Extra Bytes	Compute	Qed
0	4.7884	4.0130
100	4.8288	4.0045
1,000	4.7401	4.1144
10,000	5.1437	4.6998

Timeline Running Times (seconds)		
Extra Bytes	Compute	Qed
0	1.4835	0.5315
100	2.1761	0.6306
1,000	9.1315	0.7058
10,000	124.2409	0.8508

Tractability of the proof term is important for verification, but proof by reflection also allows the proof generation of particular lemmas using Coq's tactic system to be automatic. The tactics needed to prove a property holds for a particular disk image are simply the application of a reflection proof, a `compute`, and a `reflexivity`. This simplicity cannot be overlooked because it implies that a human operator is not needed. Let us discuss some of the further implications in the next section.

10. Future Work

We would not expect forensics analysts to prove lemmas via anything more than a reflection proof as the Coq tactic system requires a fair amount of domain-specific knowledge. On the other hand, those who are familiar with Coq might find the compute-heavy nature of these definitions to be unappealing. Compound that with the rather large (at least, for Coq) numbers needed, and it becomes safe to say that very few people would be able to use this system as it stands, 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.

The proof-by-reflection technique has shown itself quite useful in this research. However, we have largely only provided "one-way" reflection proofs; that is, that the computational version implies the `Prop` version. For example, have have shown that boolean equality between Files implies logical equality, but we have not shown whether logical equality implies the boolean form. As a matter of cleanliness, these lemmas should be extended to show that the latter also implies the former. This would prove the two versions to be equivalent, which would allow automated programs to mix and match versions as needed.

We could extend our work to include additional types of equivalences between various forms of definitions (beyond simply boolean and easy-to-read `Prop`). Providing many different (yet convertible) forms of the definitions allows our hypothetical forensics search engine to use the form which is easiest to execute or type-check. One could image, for example, a version of the byte-fetching definition that fetched bytes in chunks or one that did not construct all of the temporary structures (Inodes, Superblocks, etc.), yet could imply the definitions described in this paper. Such a version would be faster to execute but provide the same results.

Similarly, it would be helpful to create equivalences between the `File` structure and a simpler form which could more easily be manipulated by external programs/verifiers. This might be a simple list of (start, end) pairs indicating which sequences of bytes in the disk image correspond to the data contained in a file. This simpler format would allow the external proof-checkers to validate hypotheses (like `gunzip`) without needing to know anything about our `File` structures or the underlying file systems.

An external (to Coq) proof checker still needs to be written to verify hypotheses like `gunzip` and the subset relation. Such a proof checker need not be as complete as Coq, and should certainly avoid the computational limits Coq imposes. Ideally, such a checker would have a very small, hand-verified code base; it only needs to verify the types of hypotheses that Coq cannot. As we have seen with this research, this means (at minimum) the ability to read from the *whole* disk image.

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[1], that deletion events imply the associated file is marked 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. 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.

Of course, this research builds directly on the Honeynet competitions. The hackers, engineers, and researchers involved greatly advanced their field; they also provided the artifacts which served as the core scope for this paper. In particular, I should single out Matt Borland and Jason Lee, whose entries were a roadmap this research attempted to recreate.

Thanks and great appreciation also go to my partner, Laura Cathey, who has tolerated my absence far too long. Despite falling asleep while I recited this work, you helped me commit to its completion. I promise I won't start another project for at least a few days.

[1] As studied by Charles Winebrinner

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] "Chapter 8 Tactics"; section 8.7.1, `cbv`. *The Coq Proof Assistant*. `http://coq.inria.fr/refman/Reference-Manual010.html#hevea\_tactic127`

[3] Digital Forensics Framework. `http://www.digital-forensic.org/`

[4] GNOME Bugzilla. *Add support for EXIF or TIFF/EP data (in JPEG, TIFF and PNG formats*, 06/20/2001. `https://bugzilla.gnome.org/show\_bug.cgi?id=56443`

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

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

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

[8] Library of Congress. JPEG Image Encoding. *Sustainability of Digital Formats*. `http://www.digitalpreservation.gov/formats/fdd/fdd000017.shtml`

[9] "Magic Numbers". *Wikipedia: The Free Encyclopedia*. `http://en.wikipedia.org/wiki/Magic\_number\_\%28programming\%29#Examples`

[10] Minsky, Yaron and Madhavapeddy, Anil and Hickey, Jason (2013) Chapter 3. Lists and Patterns; Options. *Real World OCaml*. `https://realworldocaml.org/v1/en/html/a-guided-tour.html#options`

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

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

[13] 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`