2014 IEEE 27th Computer Security Foundations Symposium

Certificates for Verifiable Forensics

Radha Jagadeesan, CM Lubinski, Corin Pitcher, James Riely, and Charles Winebrinner
DePaul University

***Abstract*—Digital forensics reports typically document the search process that has led to a conclusion; the primary means to verify the report is to repeat the search process. We believe that, as a result, the Trusted Computing Base for digital forensics is unnecessarily large and opaque.**

We advocate the use of *forensic certificates* as intermediate artifacts between search and verification. Because a forensic certificate has a precise semantics, it can be verified without knowledge of the search process and forensic tools used to create it. In addition, this precision opens up avenues for the analysis of forensic specifications. We present a case study using the specification of a deleted file.

We propose a verification architecture that addresses the enormous size of digital forensics data sets. As a proof of concept, we consider a computer intrusion case study, drawn from the Honeynet project. Our Coq formalization yields a verifiable certificate of the correctness of the underlying forensic analysis.

I. Introduction

Eoghan Casey, editor-in-chief of *Digital Investigation*, a premier journal in computer forensics, recently editorialized [1]:

> *Digital forensics can no longer tolerate software that cannot be relied upon to perform specific functions. The root of this problem is a lack of clearly defined software requirements, which compels users and tool testers to make educated guesses and assumptions about how we expect digital forensic tools to work. This makeshift approach results in untested errors in our tools that can lead to verdicts based on incorrect information and can damage the reputation of individual practitioners and the field as a whole.*

The Scientific Working Group on Digital Evidence has raised similar concerns [2].

The complaint is not merely about the correctness of tools, it is also about underspecification. For example, consider recovery of deleted files. Precise recovery of deleted files is impossible, in general, since information has been lost. As a result, heuristics are used. Heuristics may have both false positives (incorrectly reconstructing a file) or false negatives (failing to return a deleted file). Informally, each heuristic is intended to refine the specification of file recovery only under certain assumptions.

Forensic tools that recover deleted files rarely describe the heuristics that they implement, leaving practitioners to discover and share this information. For example, Casey reports experiments demonstrating that two forensic tools differ in recovering deleted files from FAT file system images [3]. The WinHex tool attempts to recover the contents of a deleted file by taking as many contiguous clusters as needed, whether or not they are currently unallocated. The EnCase tool instead takes as many unallocated clusters as needed, but skips over currently allocated clusters. Consequently, both tools may have false positives, and they may be different! Neither tool is buggy with respect to their chosen heuristic: the problem lies with the lack of information about their chosen heuristic and the assumptions under which it is correct. [3] observes:

> *Ultimately, no single method will always be successful in all circumstances. This emphasizes the importance of using multiple tools and being aware of the assumptions they make.*

The use of forensic tools that implement complex and undocumented heuristics raises doubts about the correctness of forensic results. Practitioners do attempt to replicate results with different forensic tools, but direct comparison of results can be challenging. For example, what does it mean to say that a disk image includes evidence that a visit to an incriminating web page has been made a specific number of times? Differences in results for this query have led to disputes over the validity of digital forensic evidence in court [4].

A similar complaint can be lodged against the results obtained by composing tools. A forensics report is typically a narrative description of the forensic search: the report lists the tools that have been used and provides an argument for the conclusion based on the composition of those tools. The validity of the conclusion depends upon the details of this composition.

We see this as a problem of an overly large Trusted Computing Base (TCB). The forensic examiner relies upon tools being correct, and the court relies upon the choices, interpretations, and arguments made by the examiner. Our goals are to identify what must be trusted, and remove the need to trust wherever possible, thereby making digital forensics more trustworthy.

We are inspired by the use of certificates for SAT solvers: one should not trust the SAT solver that produces a certificate, only the verifier that checks it. In our approach, the forensic search process produces a *forensic certificate* that precisely identifies the claims and assumptions made, along with proofs that the claims follow from the assumptions.

Using the Coq theorem prover, we have formalized parts of two submissions to the Honeynet Project forensics challenge [5] as forensics certificates. We describe one of these in section V of this paper.

Specifications provide the foundation for trustworthy digital forensics, and so it is important to develop confidence in forensic specifications. In section VI of this paper we show how to analyze heuristics (encoded as specifications) for the recovery of deleted files.

Research supported by NSF 0915704 and 0916741.

© 2014, Radha Jagadeesan. Under license to IEEE.
DOI 10.1109/CSF.2014.11

IEEE computer society

The formalizations described in this paper are available at http://fpl.cs.depaul.edu/projects/forensics/.

The remainder of this section introduces the key ingredients of the forensic certificate architecture in the context of a sample investigation.

A. Borland's Report

The forensic report by Matt Borland was one of the top five submissions to a Honeynet Project forensics challenge [5]. The particular challenge was to identify a rootkit[1] within a Linux system. The evidence available to the participants was an image of the Ext2 file system. The reports for this challenge constitute informal, natural language arguments for an interpretation of data within the file system image.

Borland's challenge report states:

> *The rootkit was found on deleted inode 23 of the file system. It is a tar/gzipped file containing the tools necessary for creating a home for the attacker on the compromised system.*
>
> *Due to evidence within the installation program contained within, I will call this rootkit 'lk.tgz.' Here are its contents:*
>
> ```
> drwxr-xr-x ... 2001-02-26 14:40:30 last/
> -rwxr-xr-x ... 2002-02-08 07:08:13 last/ssh
> -rw-r--r-- ... 2001-02-26 09:29:58 last/pidfile
> -rwx------ ... 2001-03-02 21:08:37 last/install
> -rwx------ ... 2001-02-26 09:22:50 last/linsniffer
> -rwxr-xr-x ... 1999-09-09 10:57:11 last/cleaner
> ```
>
> *[...]*
>
> *I used icat to extract the first file, specifying the inode using the following commands:*
>
> ```
> $ icat honeynet/honeypot.hda8.dd 23 > recovered/file-23
> ```
>
> *Then using 'file,' I am given a guess at the type of file.*
>
> ```
> $ file recovered/file-23
> recovered/file-23: gzip compressed data, deflated,
>   last modified: Fri Mar  2 21:09:06 2001, os: Unix
> ```
>
> *At this point, I could then say:*
>
> ```
> $ tar tzvf recovered/file-23
> ```
>
> *Which then lists the contents of the tar/gzipped file, which in this case returns the listing I included in my analysis of the rootkit.*

This inclusion of tool names, input parameters and output is typical of forensic reports. In general, these tools interpret data structures or search for data. A report may not contain the entire output from the tool; a reader may choose to believe that the omitted output is irrelevant.

Borland's report uses `ils` from The Coroner's Toolkit to list the inodes of deleted files, `ils2mac` to translate the output, and `mactime` to extract access times for the inodes of deleted files.

```
# ils honeynet/honeypot.hda8.dd > ilsdump.txt
# ils2mac ilsdump.txt > deleted.txt
# mactime -p ../hp/etc/passwd -b deleted.txt 1/1/2001
```

This kind of tool composition is also typical. Here, the output of `ils` is fed to `mactime`. The examiner must also massage the data from one format to another using `ils2mac` to accommodate the ad hoc formats used by these tools.

[1] A rootkit typically contains tools used to gain and maintain administrator or system privileges on a compromised machine.

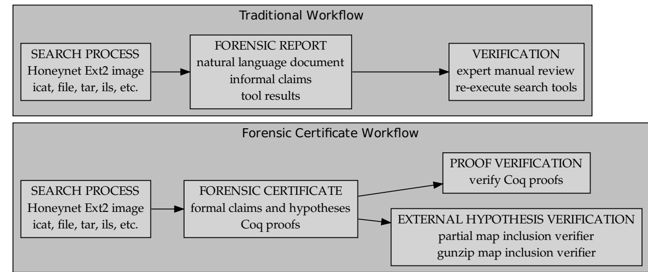


Figure 1: Traditional and Forensic Certificate Workflows

The report also includes expert interpretation. For example, the contents of the `/root/.bash_history` file suggest that new software was installed.

```
mkdir /var/...; cd /var/...
ftp ftp.home.ro
tar -zxvf emech-2.8.tar.gz
cd emech-2.8
./configure; make; make install
./mech
```

This information is used in conjunction with timestamps to form, and subsequently justify, a hypothesis about the intruder's activities:

> *...their ftp to* `ftp.home.ro` *would explain the ~7 minutes between login at 8:45 and change of* `/etc/ftpaccess` *at 8:52...*

Borland's report demonstrates the challenges involved in: (a) identifying the precise semantics of tools and natural language statements; and (b) independently verifying the behavior of tools, their composition, and arguments involving expert interpretation.

B. Formalizing Borland's Conclusion

The traditional workflow for generation and verification of forensic reports involves manual review by experts and re-execution of forensic search tools to confirm results. Figure 1 shows how our forensic certificate workflow differs from the traditional workflow. In the forensic certificate architecture, the forensic report is replaced with formal claims and Coq proofs of the claims. The proofs can be verified (without human involvement) using the Coq theorem prover. The claims in the forensic certificate may depend on hypotheses that are verified by other tools; we discuss hypotheses in subsection I-C and subsection I-D.

Borland's primary claim is that there is a deleted file on the disk that, when uncompressed, looks like a rootkit. Thus, our primary claim is a predicate, defined in Coq, that explains what has been found in a file system image:

```
Definition borland_rootkit (img : Map) : Prop :=
  exists (file : File),
```

```
isOnDisk file img
/\ isDeleted file
/\ isGzip file img
/\ Tar.looksLikeRootkit (gunzip file img).
```

This predicate indicates that (a) the `file` is found on the image `img`, (b) the `file` is deleted, (c) the contents of `file` is gzip-compressed, and (d) when decompressed, the result is a tar archive that is consistent with a rootkit. For this final predicate, Borland described the contents of the archive as containing "necessary" files for a rootkit installation; we used the list of names he provided in our formalization of `looksLikeRootkit`.

While file systems must have clear semantics for non-deleted files, heuristics for recovery of deleted files are often poorly specified, complex, or both. Fortunately, deletion in an Ext2 file system is not very destructive; the inode is simply marked as unused, and its link count zeroed. Therefore, we were able to use a very simple predicate for `isDeleted`.

The primary challenge of the formalization lies with the large volume of data common to digital forensics. The Ext2 file system image from the Honeynet challenge is 259MB. This is an unusually small file system. Nevertheless, it is too large to load into traditional theorem provers.

To gain a sense of the size limitations in existing theorem provers, consider the representation of file system images as maps using AVL trees from the Coq standard library, with binary integers for the keys and values in the map. In this representation, creating and looking up a single element in the map $\{n \mapsto n \mid 0 \leq n \leq 10^4\}$ takes 10 seconds and 400MB of RAM to compute, and the same operations for the map $\{n \mapsto n \mid 0 \leq n \leq 10^5\}$ take ~180 seconds and 1.2GB of RAM to compute.

Is it possible to identify a subset of the file system image that justifies the forensic claim and can be reasoned about using a theorem prover? And, if so, is it sound to conclude that the forensic claim holds for the entire file system image when it holds for the subset?

In many cases, the answer to both questions is yes. For example, consider a statement that a file exists within a file system image. The relevant subset of the image includes file system metadata and the sequence of directory entries leading to the file.

Below we discuss a general approach, using externally-verified hypotheses, to arguments that cannot be conducted entirely within a traditional theorem prover.

C. Making Reliance on External Verifiers Explicit

Consider a forensic claim $\phi(\mathsf{Img})$ about a large file system image Img. We wish to create and verify proofs of $\phi(\mathsf{Img})$, but Img is too large to represent in a traditional theorem prover.

The first step is to present the argument for $\phi(\mathsf{Img})$ via the following implication, for some hypothesis $\psi(\mathsf{Img})$:

$$\psi(\mathsf{Img}) \Rightarrow \phi(\mathsf{Img})$$

Since Img is too large, we cannot hope to prove this implication directly in a traditional theorem prover either. Instead, we choose the hypothesis to ensure that a traditional theorem prover can establish the statement:

$$\forall \mathsf{img}, (\psi(\mathsf{img}) \Rightarrow \phi(\mathsf{img}))$$

Here the concrete file system image Img is replaced with the quantified variable img, thereby reducing the size of the statement and its proof.

In order to conclude $\phi(\mathsf{Img})$, we rely on an *external verifier* to check $\psi(\mathsf{Img})$. For example, we make frequent use of hypotheses about whether a partial map is a subset of a file system image; it is trivial to write a program to test this property efficiently.

Importantly, the creator of a forensic certificate neither uses nor determines the external verifier implementations used by a verifying party. If an external verifier incorrectly verifies a hypothesis of the certificate, the implication of the forensic certificate is still intact, but an unwarranted conclusion might be drawn. Nevertheless, the *reliance* on a complex property is made explicit via the hypothesis. Moreover, the reliance is on a specification $\psi(\mathsf{Img})$ not a particular implementation (external verifier). This means that different verifying parties need not use the same external verifier to check $\psi(\mathsf{Img})$.

This separation is motivated by pragmatism. In practice, we expect external verifiers to be relatively simple and easy to validate, perhaps using multiple available implementations. Any more complex reasoning should be proven using the primary theorem prover.

At one extreme, if $\psi(\mathsf{Img}) = \mathsf{true}$, then the external verifier does nothing. At the other extreme, if $\psi(\mathsf{Img}) = \phi(\mathsf{Img})$, then the argument shifts entirely from the traditional theorem prover to the external verifier.

By making reliance on hypotheses explicit, we expose such tension between using complex hypotheses (and external verifiers) to simplify proofs and using simple hypotheses (and external verifiers) that are easier to trust. This provides a framework in which we can discuss simplification of hypotheses (and external verifiers). For example, to assert that a file system image contains precisely one JPEG file, it is necessary to show that every other file is not a JPEG file. An external verifier to check that a file is not a JPEG may be too complex for comfort. However, an external verifier to check that a file does not have a JPEG signature is much simpler, and can be shown to imply that a file is not a JPEG in a traditional theorem prover. Thus heuristics used by forensic search tools can be exposed and treated formally.

D. Hypotheses for Borland's Report

In the case of the Honeynet challenge, our case study shows that Borland's forensic claim can be justified using a subset of ~2,400 bytes from the file system image, declared as follows (offset 1,024 is the beginning of the superblock):

```
Definition honeynet_img_partial : Map_N_Byte := [
  1024 |-> ("216":Byte), 1025 |-> ("002":Byte), ...
]
```

Our main claim is then dependent on the assumption that

```
honeynet_img_partial ⊑ honeynet_img
```

where $\sqsubseteq$ is the subset ordering on maps represented as functions. To make use of this inclusion, the predicates in the forensic claim must be monotone with respect to $\sqsubseteq$. More complex strategies must be adopted to reason about non-monotonic predicates, e.g., "there is no file named `heist.doc`" or the more intricate "there are precisely 3 web pages in the browser cache that refer to the heist".

Similar difficulties arise with decompression. It is certainly possible to formalize `gunzip` in Coq, but in practice this is unattractive due to the volume of data involved. Instead, the main theorem uses a second map, `gunzipped_partial`, and postulates the relationship

```
gunzipped_partial ⊑ gunzip file23 honeynet_img
```

where `file23` is bound to the particular deleted inode where the rootkit was found.

Our formalization then consists of:

(a) a Coq-verified proof of the implication

```
Lemma borland_honeynet_final :
  forall (img : Map),
    honeynet_img_partial ⊑ img ->
    gunzipped_partial ⊑ gunzip file23 img ->
    borland_rootkit img.
```

(b) the claims that must be verified by external tools of the relying party's choice: (1) that `honeynet_img_partial` is a subset of the entire file system image, and (2) that `gunzipped_partial` is a subset of the content obtained by decompressing `file23`.

This provides a formal, verifiable interpretation of Borland's report. In this paper, we focus on the Coq proof for (a).

We have written simple programs to verify the claims in (b). In total the programs have fewer than 200 lines of Scala code, not including library code such as file I/O and decompression. These tools are not themselves formally verified. As we have argued in the rest of this section, these tools are not forced on the verifier of the certificate; indeed, they are free to use their own tools.

E. Contributions

In this paper, we report the following contributions:

- A general framework for the representation and verification of forensics certificates. The architecture is discussed in the next section, techniques for scaling to large data sets in section III, and generation of compact Coq proofs in section IV.
- A case study formalization of Borland's report for the Honeynet challenge, which we describe in more detail in section V.
- A novel approach to the specification and analysis of forensic data-recovery heuristics based on providing patches that restore data structures, presented in section VI.

We discuss related work in section VII.

II. Architecture for Forensic Certificates

Our approach to obtaining sound, reproducible results is based on the idea of forensic search tools producing forensic certificates that are verified by independent software. Importantly, the verification of a forensic certificate can be conducted:

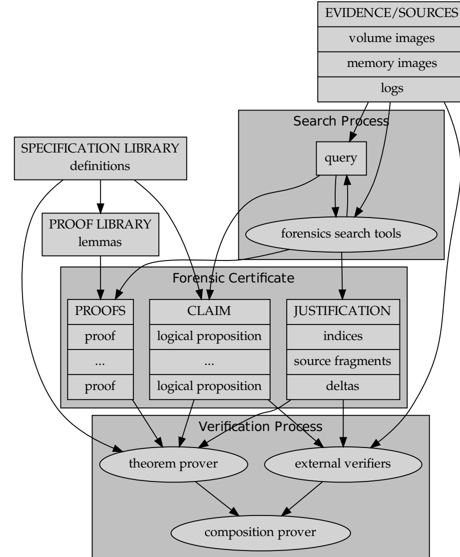


Figure 2: Forensic Certificate Architecture

- after the search has been completed, both by the creator of the certificate and by independent experts during subsequent legal proceedings,
- without access to the search software,
- with alternative specification libraries, and
- more efficiently than the original search.

The architecture is depicted in Figure 2. We discuss each major component below.

Search process: The search process operates on a collection of evidence/sources such as media images containing volumes or file systems, memory images, log files, etc. An investigator develops a query and executes search tools iteratively, refining the query based on search results.

The search process yields a claim that is supported by justifying data identified during the search, as well as proofs. We refer to the composite object as a forensic certificate. We now consider each component of the forensic certificate.

Forensic Certificate: Claim: The claim is an assertion about the evidence. We give three simple examples:

(a) *a file exists at an absolute path within a file system*
(b) *a deleted file exists within a file system*
(c) *there are no occurrences of a particular file within a file system (deleted or otherwise)*

The meaning of (a) is clear. The meaning of (b) is uncertain, because there is no canonical definition of what it means to be a deleted file in a file system. In fact, different tools use different definitions [3]. Unlike (a) and (b), (c) is a negative

assertion, and can only be validated by examining much of the file system image. However, the scope of the examination is uncertain.

We require claims to be formalized in an appropriate logic. This formalization acts as an *unambiguous interface*, and thus facilitates a clear separation between search and independent verification of results.

Forensic Certificate: Justification: A naive approach to verifying a claim is to execute a second search using alternative trusted software, and compare the results with the original claim. This is the approach taken by traditional testing, e.g., http://www.cftt.nist.gov. In addition to the obvious problems with this technique (availability of alternate tools, need to compare output with potentially different semantics, etc.), it is also inefficient. Efficiency can be improved by insisting that search software identifies the values that justify claims. For example, if a deleted file is claimed, the location of the file contents and/or a directory entry must be provided as a justifying value. Thus the problem is shifted from one of search to one of verification of a claim with justifying values.

We observe that the justifying values arise as witnesses for existential quantifiers and disjunctions in logical specifications of data structures such as file systems. For example, a logical formula for "*a file exists within a file system*" would include existentials representing the location of the file's content and the location of the directory entry for the file, amongst other metadata structures. Thus logical specifications provide a design criterion for the justifying data that is used to verify a search result. This becomes more significant as the complexity of file system structures grows.

Justifying data need not be limited to parts of the original evidence, but may also include information found during computational search. Such justifying data includes parameters such as reconstructed RAID parameters and recovered passwords / keys. More complex examples are possible. For instance, one may provide a file that is consistent with a remnant (such as a hash or file fragment) found in the evidence.

Forensic Certificate: Proofs: It is possible to write direct verifiers for families of claims and justifying values, but such verifiers inherit the complexity of the data structures (and their specifications), and there is no a priori reason to trust that they are correct. Instead we require claims to be verified using a theorem prover. Semi-automated theorem proving systems such as Coq, Isabelle, etc., allow users to prove theorems interactively. We avoid this mode of operation, and instead require that a forensic certificate includes a proof of each component in the claim, using the justifying values as witnesses. Proofs are specific to a particular theorem prover, but justifying values are not. Proofs may use lemmas from a proof library, which is not part of the TCB because the proofs in the library can be verified. We discuss methods for creating proofs in section IV.

Specification Library: The specification library provides logical specifications of data structures for, e.g., file systems, volume systems, or memory images. In particular, the specifications extend to partially-corrupted data structures, such as definitions for remnants of a deleted file existing in a file system. In such cases, there are often alternative definitions. Making these definitions precise resolves ambiguity in forensic search tools. We discuss reasoning about the correctness of definitions in the specification library in section VI.

Verification Process: Theorem Prover: The theorem prover takes as input logical formulas from the claim as well as the proofs with justifying values. The prover verifies the proof without human input. If the verification of the proof fails, the forensic certificate is rejected. Formulas within the claim can be verified independently of one another, facilitating parallel or randomized testing of proofs within a set of results where desirable. Note that the verification process does not require access to the forensic search tools used to generate the forensic certificate.

Perhaps surprisingly, the theorem prover does not use the evidence directly: existing theorem provers are unable to manage the quantity of data found in typical evidence. Instead we rely on the search process to identify fragments of the evidence that are relevant to particular propositions in a claim. The theorem prover manipulates fragments instead of the original evidence. These fragments are part of the justifying values in the forensic certificate.

Although the architecture is not tied to a particular theorem prover, we have used Coq. Thus our claims are written in a typed, higher-order logic, and proofs are represented as proof terms. We use computational reflection as a strategy for generating compact proofs. See section IV.

Verification Process: External Hypothesis Verifiers: To detect accidental or deliberate inaccuracies in the fragments, the verifier must also establish that the fragments of evidence used by the theorem prover are consistent with the original evidence. The task is ill suited to general purpose theorem provers due to the large quantity of data. We therefore verify the relationship between fragments and the original sources with separate tools. We refer to these tools as *external verifiers*, because they verify propositions, from the claim, that are hypotheses for the forensic claims verified by the theorem prover. The external verifiers make use of justifying values and the original evidence.

There are many verification tasks with similar qualities, e.g., calculating cryptographic hashes of part of a media image; scanning a media image to identify all occurrences of regular expression matches; etc. Our architecture permits the introduction of an external verifier for each task. By intent, this part of the verification is low complexity (e.g., testing whether the fragment is a contiguous subsequence of an original media image).

Verification Process: Composition Verifier: The verification of a forensic certificate is divided between the theorem prover and external verifiers for reasons of scale and efficiency. To ensure validity of the final conclusion, additional deductions must be verified. These simple first-order deductions can be verified using a theorem prover that is specialized to reason about large data. The use of a separate composition verifier avoids the need to modify the proof-checking kernel of a sophisticated theorem prover such as Coq.

Integration with Existing Forensic Search Tools: In this paper we focus on demonstrating the feasibility of the verification process and analysis of forensic definitions in the spec-

ification library. We leave integration with existing forensic search tools (creation of forensic certificates) to future work. Nevertheless, there are reasons to believe that the integration is less costly than might be expected.

The creation of logical propositions from queries is trivial engineering. The development of a specification library varies from routine to requiring considerable reverse engineering and validation, but rigorous specifications are a useful goal in their own right, and this effort can be amortized across multiple forensic search tools.

The use of proof by reflection, as discussed in section IV, shifts the burden of proof creation from forensic search tools to the developers of proof libraries that can be developed independently from forensic search tools. The Honeynet case study of section V demonstrates feasibility of constructing such proofs.

Thus the primary integration task is the generation of justifying values for the forensic certificate. It appears that in many cases of interest, simple post-processing of forensic search tool output will suffice. For example, Garfinkel proposes DFXML as a common intermediate format to facilitate composition of forensic search tools [6]. It is unsurprising that the same intermediate format also facilitates composition with verification tools. In particular, justifying values appear in the DFXML output from forensics search tools. As an example, the `fiwalk` program (integrated with the well-known Sleuthkit system) generates the following output for the rootkit file in inode 23 of the Honeynet example (some elements and attributes are omitted for brevity):

```
<fileobject>
 <parent_object> <inode>2</inode> </parent_object>
 <filename>lk.tgz</filename>
 <partition>1</partition>
 <name_type>r</name_type>
 <filesize>520333</filesize>
 <unalloc>1</unalloc> <used>1</used>
 <inode>23</inode>
 <mode>420</mode>
 <nlink>0</nlink>
 <uid>0</uid> <gid>0</gid>
 <mtime>2001-03-16T01:36:48Z</mtime>
 <ctime>2001-03-16T01:45:05Z</ctime>
 <atime>2001-03-16T01:44:50Z</atime>
 <dtime>2001-03-16T01:45:05Z</dtime>
 <byte_runs>
  <byte_run file_offset='0' fs_offset='314368' len='12288'/>
  <byte_run file_offset='12288' fs_offset='327680' len='262144'/>
  <byte_run file_offset='274432' fs_offset='591872' len='245901'/>
 </byte_runs>
 <hashdigest type='sha1'>4b0874...</hashdigest>
</fileobject>
```

There are two challenges in the use of DFXML output. The first is that there is no rigorous, formal definition of the semantics of DFXML, and different tools may use the same representation with different meanings. This is a pre-existing problem that would be detected by the verification process. The second challenge is that forensic search tools may not output all of the required justifying values, even when the search tool has the values. For example, in the process of developing a standalone verifier for DFXML output with the NTFS file system, we discovered that some required values were not included (specifically, DFXML only lists the first MFT entry for each file, but there could be many for a single file). Such missing values can usually be identified by an inexpensive, yet redundant, local search, when it is not feasible to modify the search tool to output the value.

III. Feasibility of Formal Forensics

The simplest formal model of data forensics assumes that evidence can be referenced directly in formulas used by a theorem prover. However, such a simple model is infeasible in an unmodified prover due to the size of the data to be examined. While it is possible to modify a prover to provide efficient access, this would require modifications to the trusted verifier core of existing provers, prompting soundness concerns.

Instead, as described in the previous section, we factor verification between a *base verifier* (such as Coq) and *external verifiers*. The external verifiers are trusted to approve or repudiate statements of the base logic without formal proof. The external verifiers are designed to efficiently check properties of large data sets (for example, that certain bytes are present at certain addresses). Results from the base verifier and external verifiers are designed to be easy to compose with simple first-order deduction.

A. Reasoning via Bounds

Let Img be a map from offsets to bytes, representing a file system image. Suppose a search tool identifies that property ϕ holds for Img. Let img be a fresh variable and let ϕ' be the formula obtained by replacing all occurrences of Img by img in ϕ; thus $\phi = \phi'[\mathsf{img} := \mathsf{Img}]$. Since Img is large in almost all forensic applications, the formula ϕ will be too large for existing theorem provers, even when ϕ' is small.

Many properties of interest in digital forensics do not require that the entire image be examined. For example, file identification tools often inspect only a few bytes at the start or end of a file. Similarly, affirming the existence of a file with a given pathname requires checking the bytes in a sequence of directory entries and some metadata.

We therefore adopt *partial* maps as our standard representation of disk images and seek to keep these as small as possible. We rely on forensics search tools to isolate the submap $\mathsf{Bnd} \sqsubseteq \mathsf{Img}$ that is relevant to the property of interest. The notation $b \sqsubseteq i$ indicates that when b is defined it agrees with i and that b is undefined whenever i is undefined. The proof for ϕ then consists of the following three components (recall that $\phi = \phi'[\mathsf{img} := \mathsf{Img}]$):

$$\forall \mathsf{img}.\ (\mathsf{Bnd} \sqsubseteq \mathsf{img} \Rightarrow \phi') \tag{1}$$

$$\mathsf{Bnd} \sqsubseteq \mathsf{Img} \tag{2}$$

$$\{\forall, \Rightarrow\}\text{-elimination to conclude } \phi \tag{3}$$

If both Bnd and ϕ' are small, then (1) is small, and therefore it can be checked by the base verifier. Since (2) includes Img, this formula must be checked by an external assumption verifier. Trust in the external verifier must be established separately, but the decision procedure for map inclusion is extremely simple. The deduction (3) is a first-order deduction carried out by the composition verifier. That is, $\forall$-elimination is used to deduce $(\mathsf{Bnd} \sqsubseteq \mathsf{img} \Rightarrow \phi')[\mathsf{img} := \mathsf{Img}] = (\mathsf{Bnd} \sqsubseteq \mathsf{Img} \Rightarrow \phi)$ from (1) by instantiating the variable img with the file system image Img. Then, $\Rightarrow$-elimination (modus ponens) is applied to this result, along with (2), to deduce (3).

Monotonicity: Often, formula (1) above can be established by demonstrating that ϕ' is monotone with respect to $\sqsubseteq$ and then establishing $\phi'[\mathsf{img} := \mathsf{Bnd}]$. That is, the following statements are established in the base verifier, then (1) is deduced.

- Monotonicity of ϕ':

$$\forall \mathsf{img}_1, \mathsf{img}_2.\ (\mathsf{img}_1 \sqsubseteq \mathsf{img}_2 \land \phi'[\mathsf{img} := \mathsf{img}_1] \Rightarrow \phi'[\mathsf{img} := \mathsf{img}_2])$$

- ϕ' is shown to hold for Bnd: $\phi'[\mathsf{img} := \mathsf{Bnd}]$

Such monotonicity properties are reminiscent of those that arise in model theory for positive existential formulas.

Example III.1 (Monotonicity of Parsing Ext2 Superblocks). Monotonicity results are often obtained compositionally, and the intermediate results may have additional quantification. For example, the procedure to compute a superblock record from an Ext2 file system image is monotone in the image. This is captured in the following Coq lemma from our Honeynet development (where the `Found` constructor indicates a successful operation):

```
Lemma findAndParseSuperBlock_subset :
  forall (img1 img2 : Map) (superblock : SuperBlock),
    img1 ⊑ img2 ->
      findAndParseSuperBlock img1 = Found superblock ->
        findAndParseSuperBlock img2 = Found superblock.
```

B. Spatial Decomposition

A search for documents in a file system often produces thousands of files. In such cases, the claim for a forensics search yields a formula that is too large to represent in existing theorem provers such as Coq. Fortunately, forensics search results are often naturally represented as conjunctions of subformulas that refer to distinct areas of an image. Subformulas are checked by the base verifier, and conjunctions of those subformulas are checked by the simpler composition verifier, facilitating, e.g., parallel or randomized verification of components of forensics results.

Consider a statement $\phi \triangleq \bigwedge_{i \in I} \phi_i$, where Img appears in each subformula ϕ_i. As an example of such a statement, each ϕ_i may describe the existence of a file in Img. Assume a family of partial maps $(\mathsf{Bnd}_i \mid i \in I)$ and formulas $(\phi'_i \mid i \in I)$ such that $\phi_i = \phi'_i[\mathsf{img} := \mathsf{Img}]$, each ϕ'_i contains no occurrences of Img, and each ψ_i is provable with the base verifier, where:

$$\psi_i \triangleq \forall \mathsf{img} : \mathsf{Map}.\ (\mathsf{Bnd}_i \sqsubseteq \mathsf{img} \Rightarrow \phi'_i) \qquad (\forall i \in I)$$

Now the size of the conjunction $\bigwedge_{i \in I} \psi_i$ may still be too large for the base verifier because of the size of $\Sigma_{i \in I} |\mathsf{Bnd}_i|$. For example, if ϕ_i represents a statement about the contents of a file in a file system, then each Bnd_i could be the entire file contents. Moreover, if I ranges over all files within a file system, then the size of the statement $\bigwedge_{i \in I} \psi_i$ is linear in the size of the allocated space within the file system. For this reason, the final deduction of ϕ must be performed by the composition verifier.

The proof for ϕ then consists of three components:

$$\forall \mathsf{img}.\ (\mathsf{Bnd}_i \sqsubseteq \mathsf{img} \Rightarrow \phi'_i) \qquad (\forall i \in I) \tag{4}$$

$$\mathsf{Bnd}_i \sqsubseteq \mathsf{Img} \qquad (\forall i \in I) \tag{5}$$

$$\{\forall, \Rightarrow\}\text{-elimination and } \{\land\}\text{-introduction to conclude } \phi \tag{6}$$

Again (4) is checked by the base verifier, (5) by an external verifier, and (6) by the composition verifier.

More generally, it may not be possible to find small partial maps as above. For example, consider the statement that a directory exists at an absolute path $/d_0/d_1/\ldots/d_{N-1}$. If ϕ'_i states that there is a directory entry with name d_i and a path from the root directory, then establishing ϕ'_i requires additional bounds on img:

$$\forall \mathsf{img}.\ (\bigwedge_{0 \le j \le i} \mathsf{Bnd}_j \sqsubseteq \mathsf{img}) \Rightarrow \phi'_i) \qquad (\forall i \in I) \tag{7}$$

However, $\Sigma_{0 \le j \le i} |\mathsf{Bnd}_j|$ may be too large. Instead, we establish ϕ as above, replacing (4) with:

$$\forall \mathsf{img}.\ (((\bigwedge_{0 \le j \le i-1} \phi'_j) \land \mathsf{Bnd}_i \sqsubseteq \mathsf{img}) \Rightarrow \phi'_i) \qquad (\forall i \in I) \tag{8}$$

Note that the upper bound on j is $i-1$, rather than i, and that the ith formula in (8) contains exactly 1 partial map (Bnd_i) rather than the $i+1$ partial maps in the subformula $(\bigwedge_{0 \le j \le i} \mathsf{Bnd}_j \sqsubseteq \mathsf{img})$ of (7). Perhaps unusually, this application of cut serves to reduce the size of the formulas under consideration rather than the size of derivations.

C. Sound and Complete Search Results

Consider a forensics search for a set of offsets X satisfying a property P, e.g., occurrences of regular expression matches. Here we consider proofs that:

(a) each offset in X satisfies P (soundness), and that
(b) X contains every offset satisfying P (completeness).

Soundness: We must establish that each offset in X satisfies P. Let $\mathsf{dom}(\mathsf{Img}) = [0, N)$ for some N. The soundness of the search can be expressed as follows.

$$\phi \triangleq \forall n, 0 \le n < N \Rightarrow (n \in X \Rightarrow P(\mathsf{Img}, n))$$

As before, we assume an indexed collection of partial maps $(\mathsf{Bnd}_n \mid n \in X)$ such that the following is provable with the base verifier:

$$\forall \mathsf{img}.\ \mathsf{Bnd}_n \sqsubseteq \mathsf{img} \Rightarrow P(\mathsf{img}, n) \qquad (n \in X)$$

Now $\Sigma_{n \in X} |\mathsf{dom}(\mathsf{Bnd}_n)|$ may be large, so the base verifier cannot generally deduce:

$$\forall \mathsf{img}.\ (\bigwedge_{n \in X} (\mathsf{Bnd}_n \sqsubseteq \mathsf{img})) \Rightarrow \\ \forall n, 0 \le n < N \Rightarrow (n \in X \Rightarrow P(\mathsf{img}, n))$$

Instead, we use the base verifier to establish that individual results about each Bnd_n, for $n \in X$, can be combined using formula (10) below. The complexity of (10) arises from the quantification over bounds used for partial maps (eliminated in the subsequent composition verifier deduction). The complexity is justified by the reduction in size of the formula presented to the base verifier, and by the simplicity of the composition verifier. The proof for soundness has the following steps:

$$\forall \mathsf{img}.\ \mathsf{Bnd}_n \sqsubseteq \mathsf{img} \Rightarrow P(\mathsf{img}, n) \qquad (n \in X) \tag{9}$$

$$\begin{array}{l}\forall(\mathsf{bnd}_n \mid n \in X). \\ \quad (\bigwedge_{n\in X}(\forall \mathsf{img}.\ \mathsf{bnd}_n \sqsubseteq \mathsf{img} \Rightarrow P(\mathsf{img},n))) \Rightarrow \\ \quad \forall \mathsf{img}.\ ((\bigwedge_{n\in X} \mathsf{bnd}_n \sqsubseteq \mathsf{img}) \Rightarrow \\ \qquad\qquad (\forall n.\ n \in X \Rightarrow P(\mathsf{img},n)))\end{array} \tag{10}$$

$$\mathsf{Bnd}_n \sqsubseteq \mathsf{Img} \qquad (n \in X) \tag{11}$$

$$\{\forall, \Rightarrow\}\text{-elimination and } \{\wedge\}\text{-introduction to conclude } \phi \tag{12}$$

Here (9) and (10) can be checked by the base verifier, (11) by the external verifier, and (12) by the composition verifier. Note that the universally-quantified variables $(\mathsf{bnd}_n \mid n \in X)$ in (10) are instantiated with the partial maps $(\mathsf{Bnd}_n \mid n \in X)$ in the composition verifier.

Completeness: Now we must establish that every offset satisfying P appears in X:

$$\phi \triangleq \forall n.\ 0 \leq n < N \Rightarrow (P(\mathsf{Img},n) \Rightarrow n \in X)$$

Since the entirety of Img must be examined in general, e.g., for regular expression matches, it is unlikely that small partial maps can be used directly for such proofs. Instead we consider how to use assumptions about intervals where the property P does not hold. This is useful when the number of intervals is significantly smaller than the total size of the intervals. The assumptions must be checked using external verifiers that understand P.

To illustrate, consider a family of pairs $I \subset \mathbb{Z} \times \mathbb{Z}$ representing the intervals where P does not hold, i.e.:

$$\{n \mid (x,y) \in I \wedge x \leq n < y\} = \mathsf{dom}(\mathsf{Img}) \backslash X$$

We use trusted external verifiers to check the family of results:

$$\forall n.\ x \leq n < y \Rightarrow \neg P(\mathsf{Img},n) \qquad ((x,y) \in I)$$

Then the proof for completeness consists of:

$$\begin{array}{l}\forall \mathsf{img}.\ (\bigwedge_{(x,y)\in I}(\forall n.\ x \leq n < y \Rightarrow \neg P(\mathsf{img},n))) \\ \qquad \Rightarrow \forall n.\ 0 \leq n < N \Rightarrow (P(\mathsf{img},n) \Rightarrow n \in X)\end{array} \tag{13}$$

$$\forall n.\ x \leq n < y \Rightarrow \neg P(\mathsf{Img},n) \qquad ((x,y) \in I) \tag{14}$$

$$\{\forall, \Rightarrow\}\text{-elimination and } \{\wedge\}\text{-introduction to conclude } \phi \tag{15}$$

(13) is checked by the base verifier, (14) by an external verifier, and (15) by the composition verifier. The proof of (13) uses the fact that:

$$\forall n.\ 0 \leq n < N \Rightarrow n \in X \vee (\bigvee_{(x,y)\in I} x \leq n < y).$$

Example III.2 (Counting Deleted Files). Consider a forensic claim that there is exactly one deleted file in an Ext2 file system image. To make "deleted" precise, we mean that there is an inode with a link count of zero [7]. The link count is stored in 2 bytes within an inode. The Ext2 Honeynet image has 66,264 inodes, and so the naive formalization that uses a subset of the file system image with all inode link counts requires a partial map with 132,528 entries, which is large enough to be problematic in a theorem prover.

Fortunately, inodes are stored consecutively within a small number of block groups, and this can be used to reduce the size of the partial map used within a theorem prover. First, define the predicate $\phi(\mathsf{img},s,m,n,v,X)$ so that X is the set of offsets in s, $s+m$, $s+2m$, $\ldots$, $s+(n-1)m$, where a value v occurs in two bytes within img. That is:

$$\begin{array}{l}\phi(\mathsf{img},s,m,n,v,X) \triangleq \\ \quad X = \{\, j \mid 0 \leq j < n \\ \qquad\qquad \wedge\, \mathsf{img}[s+j*m] + 256*\mathsf{img}[s+j*m+1] = v\}\end{array}$$

A simple external verifier can be constructed to verify ϕ.

For the Honeynet challenge image there are 2,008 inodes per block group, and suppose that s_0, s_1, s_2, $\ldots$, s_{32} are the offsets for the inode tables within each block group. There are 128 bytes per inode, and the link count is stored in offsets 26-27 within an inode. If the conclusion of the forensic claim is that the deleted file occurs in the 20th inode of block group zero, the hypothesis used within the base verifier is then:

$$\begin{array}{l}\phi(\mathsf{img},s_0+26,128,2008,0,\{19\}) \wedge \\ \phi(\mathsf{img},s_1+26,128,2008,0,\{\}) \wedge \\ \phi(\mathsf{img},s_2+26,128,2008,0,\{\}) \wedge \\ \ldots \\ \phi(\mathsf{img},s_{32}+26,128,2008,0,\{\})\end{array}$$

With the above hypothesis, the definition of ϕ, and file system metadata from the superblock, it is possible to establish that there is exactly one deleted file. Note that the conclusion of the forensic claim is formulated in terms of the Ext2 file system structure. In contrast, the hypothesis does not require any knowledge of file system structure, i.e., the theorem prover verifies arguments about the interpretation of data in the file system. With this approach, the formula (an implication) checked by the theorem prover is linear in the number of block groups (33 in the Honeynet example), as opposed to being linear in the number of inodes (66,264 in the Honeynet example) by the large partial map. □

D. Composition Verifier

The arguments above, in (3), (6), (12) and (15), demonstrate that, by design, results from the base verifier and external verifiers can be composed using intuitionistic propositional reasoning and a restricted form of ∀-elimination that allows top-level quantification over images. In some cases, these deductions are simple enough that mechanization adds little. This is arguably true for the case study presented in section V, which requires one application of ∀-elimination and two applications of ⇒-elimination.

For more complex uses of propositional connectives, mechanization may be desirable. Such a composition verifier would be quite simple — much simpler than the kernel of a theorem prover for a more sophisticated logic. The only interesting requirement is that the composition verifier support the large maps arising from file system images, memory images, etc. For example, formulas might refer to the names of files containing the large maps, as opposed to having the large maps embedded within the formulas.

IV. Generation of Proofs

An important design principle for the architecture is that generation of forensic certificates must be automated, and must not require additional effort by forensic examiners. Forensic certificates contain justifying data, claims, and proofs. Existing forensic search tools often produce enough information to create the first two via an automated post-processing step, given sufficient knowledge of the search tools' behaviour. Formal proofs however are more difficult to recreate from the output of forensic search tools. In addition, the size of proofs must be minimized, as discussed in section III. Here we illustrate the issues using the Coq theorem prover, and outline how to generate compact proofs using a proof by reflection strategy.

Coq is a semi-automated type-checker for a powerful dependently-typed programming language, and is used as a theorem-proving framework. Although proof terms in Coq can be written directly, they are often constructed using a tactic language. A proof term constructed by tactics is type-checked, or verified, and can be saved for subsequent standalone verification.

Thus there are two competing methods for distributing proofs. The first is to distribute the tactics script, which is often easier to understand or modify than a proof term; however, verifying the script is potentially expensive because it entails repeating the same proof search, and tactics are less robust than proof terms because of the potential for changes in the proof search algorithms. The second method is to distribute the proof term. For the content of forensic certificates, the second method is preferable because of the potential inefficiency and non-robustness of tactic scripts.

It remains to generate the proof term to be stored in the forensic certificate. One approach is to develop custom tactics that are specific to the claims produced by forensic search tools. The custom tactics conduct an automated proof search on behalf of the forensic examiner. Following the original design principle, custom tactics are expected to find a proof if one exists, so that the forensic examiner need not be involved with the theorem prover.

Our experiments suggest that proof search for forensic claims is challenging to execute efficiently, and easily leads to proof terms that are too large to type check with Coq. In particular, the size of proof terms can be proportional to the number of cases in a proof, and naive proofs about forensic claims do generate very large numbers of cases.

A. Proof by Reflection

Rather than use tactics based search, we have adopted a *proof by reflection* strategy for the generation of proof terms. Proof by reflection [8] replaces a proof search with a computational procedure, when the computational procedure has been proven correct.

Proof by reflection relies on the following type inference rule. It allows a term `M` to be type checked with type `U` if `M` has type `T` *and* the types `T` and `U` are convertible.

$$\frac{\Gamma \vdash M : T \qquad T \equiv U}{\Gamma \vdash M : U}$$

The convertibility relation includes computation. For example, consider a boolean-valued function `f`. The previous typing rule can be instantiated to show that the proof term for reflexivity of equality `eq_refl` has type $\texttt{f x} = \texttt{true}$ whenever $\texttt{true} \equiv \texttt{f x}$, i.e., when executing `f x` results in `true`.

$$\frac{\Gamma \vdash \texttt{eq_refl} : \texttt{true} = \texttt{true} \qquad (\texttt{true} = \texttt{true}) \equiv (\texttt{f x} = \texttt{true})}{\Gamma \vdash \texttt{eq_refl} : \texttt{f x} = \texttt{true}}$$

If it is possible to deduce a property `P x` from the computation of `f x` returning `true`, then `f` can be seen as certified code for the property `P`. This would be established as a lemma `sound` of the following form:

$$\Gamma \vdash \texttt{sound} : \forall \texttt{x}, \texttt{f x} = \texttt{true} \rightarrow \texttt{P x}$$

If the reverse implication holds, `f` is a decision procedure for `P`.

If there is a lemma `sound` with type as above, then a proof of `P v`, for some literal value `v`, has a simple tactics script of the following form. It uses the `sound` lemma, and then checks that computing `f v` yields `true`.

```
Lemma result : P v.
Proof. apply sound; reflexivity. Qed.
```

B. Application to Forensic Certificates

In the context of forensic certificates with large sources and the limitations of existing theorem provers, the size of the tactics script is less important than the size of the corresponding proof term that it creates. In the case of `result` above, the proof term is:

```
sound v (@eq_refl bool true)
```

In particular, for forensic claims, the literal value `v` may be a large fragment of a disk image. Naive proof terms generated by custom tactics can have size that is quadratic or worse in the size of `v`, and so the single occurrence of `v` in the proof term above is a significant improvement (recall from section I that a single occurrence of a map with 10^5 elements uses 1.2GB of RAM in the Coq process).

In addition to decreasing the size of proof terms, our experiments suggest that it is easier to develop computational procedures than custom tactics for proving forensic claims. Moreover, the procedures are for verification of justifying data from a forensic certificate, as opposed to searching for the same data, and so the procedures are simpler than forensic search tools. However, the procedures to be developed even for simple file system claims have more variety than the code found in a file system driver, because they may verify properties of partially-corrupted data structures, e.g., for verifying claims about deleted files.

The correctness of the computational procedures is established by lemmas such as `sound` above. This allows the specification for forensic claims to be defined declaratively (e.g., via a graph logic statement encoded in Coq) rather than via a reference implementation. The former is more convenient for the type of analysis discussed in section VI.

Finally, the use of computational procedures for verifying forensic certificates facilitates reasoning about the completeness of the procedure, as opposed to custom tactics. This can

remove the risk that a forensic search produces a claim that cannot be proven despite testing of the process to generate a proof.

C. Runtime Performance

To verify each proof in a forensic certificate, the theorem prover must re-execute the computational procedure. For this reason, we must pay attention to the efficiency of reduction in theorem provers. Our experiments in this area suggest that, with care, the Coq theorem prover can perform computations for our formalization of a forensics challenge from the Honeynet project within a few seconds. In section VII we discuss related work on increasing runtime performance necessitated by the use of a proof by reflection style.

V. Case Study: Computer Intrusion

In this section, we return to our formalization of Matt Borland's forensic report for one of the Honeynet Project's challenges [5]. The formalization involves verification of results using both Coq and external verifiers, in addition to deduction between the components as discussed in section III. The following discussion reflects the fact that almost all of the development effort lay in the specification and verification using Coq.

The Coq development for this case study is available at http://fpl.cs.depaul.edu/projects/forensics/.

A. Forensic Claim

Recall the Coq definition of the predicate for the claim of Borland's report:

```
Definition borland_rootkit (img : Map) : Prop :=
  exists (file : File),
    isOnDisk file img
    /\ isDeleted file
    /\ isGzip file img
    /\ Tar.looksLikeRootkit (gunzip file img).
```

This predicate indicates that there is a deleted `file` within the file system image `img`, it is a gzip-compressed file, and the result of decompressing the file is a tar archive that is consistent with a rootkit.

The `File` type represents metadata about the file, including its size, the location of the file's content, and whether or not the file is deleted. The `isOnDisk` predicate establishes that the metadata given in `file` exists in `img`. The definition of the `isOnDisk` predicate is specific to the Ext2 file system. We leave the modular specification of other file systems to future work.

The `isGzip` predicate is a simple signature-based check of the contents of the file. This predicate examines the actual file content from `img`, where the location of the content is determined by `file`. The decision to refer to file content by indirection to `img`, as opposed to storing a copy of the file content in `file`, makes it easier to avoid duplication of the content in the resulting proof term's witness for the existential.

The `Tar.looksLikeRootkit` predicate examines the structure of a tar file, and tests whether two or more names of archive entries are on a blacklist, reflecting Borland's discussion. Borland's report does not contain an analysis of the malicious executables within the tar file. We discuss the use of `gunzip` below.

B. Proof Outline

If there were no limits on the size of data that could be handled by Coq, we would like to prove:

```
Lemma attempt1 : borland_rootkit honeynet_img.
```

where `honeynet_img` is a data structure holding the entire 259MB file system image. However, as discussed in the introduction, this data structure is many orders of magnitude larger than Coq can handle. Our second attempt is to identify a subset `honeynet_img_partial` of `honeynet_img`, and prove:

```
Lemma attempt2 : borland_rootkit honeynet_img_partial.
```

This is useful in the presence of a monotonicity result of the form (recall that $\sqsubseteq$ is the inclusion order on maps represented as functions):

```
Lemma borland_rootkit_mono :
  forall (img1 img2 : Map),
    img1 ⊑ img2 ->
    borland_rootkit img1 ->
    borland_rootkit img2.
```

Assuming a proof term `inclusion` for the order:

```
inclusion : honeynet_img_partial ⊑ honeynet_img
```

we would then have the following proof term for the original lemma `attempt1` above:

```
borland_rootkit_mono
  honeynet_img_partial
  honeynet_img
  inclusion
  attempt2
: borland_rootkit honeynet_img.
```

Since `honeynet_img`, and consequently `inclusion`, cannot be defined in Coq, this final conclusion must be checked by a composition verifier rather than Coq. Additionally, the proof of `honeynet_img_partial` $\sqsubseteq$ `honeynet_img` involves running a simple external verifier that has access to the partial image used in Coq and the full file system image.

C. Decompression

It remains to prove `attempt2`. This proof is complicated by the presence of decompression in `borland_rootkit` (the forensics claim). There are two issues. The first is that a Coq coding of the gzip decompression algorithm is non-trivial to develop. The second is that the decompression algorithm uses the entire compressed input to produce the entire decompressed output, and the entire compressed input file is also too large to represent in Coq. This is unfortunate because the predicate `Tar.looksLikeRootkit` requires only part of the decompressed file, by analogy with the use of `honeynet_-img_partial` above. It is possible to describe relationships between partial input and partial output for the decompression algorithm, but these relationships are particularly complex.

For these reasons, we rely on an external program to decompress the file. The external decompressor becomes part

of the TCB, in addition to the Coq type checking kernel, the specification library, and other external verifiers. While not ideal, including decompression in the TCB seems to be a benign compromise: we believe that errors in forensic claims are more likely to be associated with specialized, domain-specific tools and arguments rather than with standard decompression tools.

In addition to executing the decompression algorithm outside Coq, we also leave the decompression algorithm uninterpreted within the Coq specification and proof of the forensic claim. That is, the Coq script includes the declaration of a free variable for the decompression function:

```
Variable gunzip : File -> Map -> File.
```

Note that the `File` type may optionally include file content, and this is important for `gunzip` because the result of decompressing cannot normally be found in the file system image.

We have already seen that the uninterpreted `gunzip` function appears in the forensic claim `borland_rootkit`. In order to establish a property of the results of `gunzip`, we require a hypothesis about `gunzip` in the forensic claim. The hypothesis is that there is a subset `gunzipped_partial` of the decompressed data obtained from the compressed file `file23` (the file for inode 23 in the Honeynet challenge image) within the file system image, written as follows:

```
gunzip_inclusion :
  gunzipped_partial ⊑ gunzip file23 honeynet_img
```

The proof of `borland_rootkit` is then based on factoring it into the following lemma about the subsets `honeynet_-image_partial` and `gunzipped_partial` of the file system image and the decompressed file respectively:

```
Lemma attempt3 :
  isOnDisk file23 honeynet_img_partial
  /\ isDeleted file23
  /\ isGzip file23 honeynet_img_partial.
  /\ Tar.looksLikeRootkit gunzipped_partial.
```

Monotonicity results are used to extend the above lemma to larger file system images. For the first three predicates, monotonicity results resemble `borland_rootkit_mono`. For the last predicate, we show that `Tar.looksLikeRootkit` is monotone in its `File` argument:

```
Lemma looksLikeRootkit_mono :
  forall (f1 f2 : File),
    f1 ⊑ f2 ->
    Tar.looksLikeRootkit f1 ->
    Tar.looksLikeRootkit f2.
```

The individual monotonicity results are combined as:

```
Lemma borland_rootkit_mono_revised :
  forall (img1 img2 : Map) (file f1 f2 : File),
    img1 ⊑ img2 ->
    f1 ⊑ f2 ->
    (isOnDisk file img1
     /\ isDeleted file
     /\ isGzip file img1
     /\ Tar.looksLikeRootkit f1) ->
    (isOnDisk file img2
     /\ isDeleted file
     /\ isGzip file img2
     /\ Tar.looksLikeRootkit f2).
```

Outside Coq, this monotonicity result can be used with the large `honeynet_image` to create the following proof term:

```
borland_rootkit_mono_revised
  honeynet_img_partial honeynet_img
  file23
  gunzipped_partial (gunzip file23 honeynet_img)
  inclusion gunzip_inclusion
  attempt3 :
(isOnDisk file23 honeynet_img
 /\ isDeleted file23
 /\ isGzip file23 honeynet_img
 /\ Tar.looksLikeRootkit (gunzip file23 honeynet_img)).
```

Within Coq, we instead finalize the formalization of the forensic claim as:

```
Lemma borland_honeynet_final :
  forall (img : Map),
    honeynet_img_partial ⊑ img ->
    gunzipped_partial ⊑ gunzip file23 img ->
    borland_rootkit img.
```

Then the conclusion `(borland_rootkit honeynet_image)` depends only on external verification of the two properties:

```
inclusion :
  honeynet_img_partial ⊑ honeynet_image
gunzip_inclusion :
  gunzipped_partial ⊑ gunzip file23 honeynet_img
```

As discussed in section I, we have implemented external verifiers for these properties in fewer than 200 lines of Scala code.

D. Coq Development

The Coq development for Borland's report has two components:

(a) The specification of the forensic claim, including the definition of data structures for the Ext2 file system.
(b) Procedures used in proof by reflection, and proofs of lemmas.

Code in (a) is part of the TCB, in addition to the Coq standard library definitions and the Coq type checking kernel, and is approximately 800 lines of code. Code in (b) is not part of the TCB, and is approximately 1,800 lines of code.

The proof of concept currently lacks integration with existing forensics search tools. For example, we have used simple custom tools to extract `honeynet_image_partial` and `gunzipped_partial` for insertion into Coq statements. These custom tools are not part of the TCB.

Verifying the entire development, including both the forensic claim and proofs of correctness, takes approximately 26 seconds. We use Coq's `vm_compute` tactic to normalize terms with the virtual machine, as opposed to the default normalization mechanism, for reasons of speed. The default normalization mechanism requires 267 seconds to complete the proof for `attempt3`, whereas the virtual machine mechanism completes the same proof in 4.6 seconds. The reported times are for Coq 8.4pl2 and Linux kernel 3.2.0, executing on a Xeon E3-1230 running at 3.20GHz with 16GB of RAM.

VI. Analysis of Forensics Specifications

It is relatively straightforward to specify the interpretation of the current contents of a file system image. We refer to such a specification as the *standard specification*. The standard specification can be validated against a running implementation. For example, one can easily test that the specified contents of a directory are the same as the interpretation under a given file system driver.

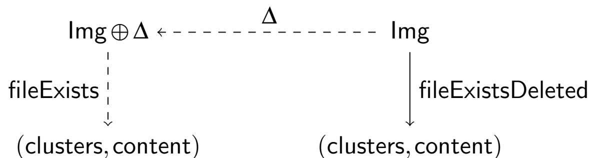
Forensics is also concerned with inferring the *past* contents of a file system. Assuming that backups are not available, such past properties must be inferred from the current contents of the file system; we refer to the specification of such past properties as a *nonstandard specification*. A nonstandard specification typically interprets data structures in the file system that are partially destroyed. For example, when a file is deleted, only part of the directory entry may be deleted by a particular implementation. In this case, it may be impossible to validate the nonstandard specification against a running implementation: the file system driver will likely ignore the corrupted entry. Other examples of nonstandard specifications are found in memory forensics and when data has been deliberately hidden by anti-forensics tools [9].

We now consider how to validate nonstandard specifications. The validation is intended to convince the relying parties that recovered data has not been fabricated by the search process. We describe our approach in subsection VI-A and present a case study in subsection VI-B. The case study formalizes reasoning about deletion in the FAT filesystem. We refer to this example informally in subsection VI-A. It is sufficient to note that when a FAT file is deleted, the list of allocated clusters is lost — only the first cluster and file length are known. For a deleted file containing n clusters, only one is known; the other $n-1$ clusters could have been anywhere on the disk, in any order.

A. Validation of Nonstandard Specifications via Patches

Since there are many possible nonstandard specifications for recovery of information from deleted files, or more generally undefined and invalid data structures, how do we know whether a particular choice of specification is reasonable? We consider *patches* to an image. A patch is intended to restore an undefined or invalid data structure to a valid state. Patches provide a deterministic, concrete explanation of changes.

We represent a patch as a partial map Δ from offsets to bytes, and write $\mathsf{Img} \oplus \Delta$ for the partial map where Δ updates Img. If a nonstandard specification justifies recovery of a deleted file from Img, does there exist a partial map Δ such that $\mathsf{Img} \oplus \Delta$ satisfies the standard specification? In the case of file deletion, let $\mathsf{fileExists}$ be the standard specification and $\mathsf{fileExistsDeleted}$ be the nonstandard specification. Diagrammatically, we have the following.



Without any constraint, of course, patching is too powerful to be useful. The entirety of the disk can be overwritten! Constraints can be used to establish that a patch is reasonable.

The simplest form of constraint limits the size of a patch. For example, a patch for the recovery of a deleted (but not overwritten) JPEG image file would be suspicious if it modified more than a few hundred bytes. Nevertheless, constraints on patch size may not identify patches that modify, e.g., EXIF metadata in a JPEG image file (to suggest that a photo was taken at a particular location and time), or the contents of a spreadsheet.

For this reason, it is important to understand the contents of a patch in terms of file system semantics. Often the domain of a patch determines whether it is acceptable. In the case of FAT file deletion, the patch modifies only one byte of the directory entry and several bytes in the FAT; only metadata is altered, not data. This is sufficient to argue that the contents of the file has not been manufactured by the forensics process. In the next subsection, we provide a more formal account.

B. Case Study: Deleted Files in the FAT File System

The FAT file system stores the contents of files in cluster chains, consisting of one or more clusters that may be contiguous or fragmented across the file system [10]. The File Allocation Table (FAT) is a data structure that records both the allocation status of clusters and the order of clusters that make up a file. Conceptually, the FAT is a partial map $\mathsf{FAT} : \mathbb{Z} \rightharpoonup \{\mathsf{UN}, \mathsf{TM}\} \cup \mathbb{Z}$. For a cluster number n, if $\mathsf{FAT}(n) = \mathsf{UN}$, then cluster n is unallocated. If $\mathsf{FAT}(n) = n' \in \mathbb{Z}$, then cluster n is allocated and it is followed by cluster n' in a cluster chain. If $\mathsf{FAT}(n) = \mathsf{TM}$, then cluster n is allocated and is the terminal entry in a cluster chain. A FAT should not have occurrences of UN in cluster chains. A directory entry in a FAT file system image contains the filename, the file size, and the starting cluster number. A file system driver accesses a file's contents by calculating the cluster chain from the FAT.

Typically, when a file is deleted, the first byte of the filename is overwritten with a distinguished value (marking the directory entry as unallocated), and the entire cluster chain is marked as unallocated in the FAT. A side effect of the FAT representation is that the cluster chain is overwritten by the deallocation.

A nonstandard specification for the existence of a deleted file in a FAT file system then has to identify: (a) an unallocated directory entry, including file size; and (b) a cluster chain storing the file contents.

It is reasonable to justify (a) either by exhibiting a path from the root directory to the directory entry, or by showing that the contents matches the form of a directory entry.

For (b), Carrier describes two simple heuristics for choosing a cluster chain for deleted files [7], based on the forensics tools reviewed in [3] (see section I). To see the heuristics for a deleted file, consider a cluster chain map $\mathsf{cluster} : [0, \mathsf{L}) \to \mathbb{Z}$ of length $\mathsf{L} \triangleq \lceil(\mathsf{fileSize}/\mathsf{clusterSize})\rceil$. The $\mathsf{fileSize}$ is stored in the directory entry and is not destroyed when the file is deleted. We insist that $\mathsf{cluster}(0) = \mathsf{init}$ is the starting cluster number given in the directory entry for non-empty files, and

that all clusters in the chain are unallocated:

$$\forall 0 \leq i < \mathsf{L}.\ \mathsf{FAT}(\mathsf{cluster}(i)) = \mathsf{UN}$$

The specification for the first heuristic states that the cluster chain is contiguous:

$$\forall 0 \leq i < \mathsf{L}.\ \mathsf{cluster}(i) = \mathsf{init} + i$$

The second heuristic generalizes the first. The specification allows the cluster chain to be fragmented, but each fragment must be filled with allocated clusters:

$$\begin{aligned}&\forall 0 \leq i < \mathsf{L}.\\ &\quad \mathsf{cluster}(i) = \\ &\qquad \mathsf{init} + i + |\{j \mid 0 \leq j < \mathsf{cluster}(i) \land \mathsf{FAT}(\mathsf{init} + j) \neq \mathsf{UN}\}|\end{aligned}$$

Coq Development: Our Coq development (approximately 9,700 lines of Coq code) provides standard and nonstandard specifications for files in the FAT16[2] file system. We prove that nonstandard specifications corresponding to the heuristics above can be represented as patches to the file system image; and that the domains of those patches are restricted to changing the allocation status of the directory entry and the FAT. These constraints on patches ensure that the directory entry is not modified to point to a different file, file content is not modified, etc.

We represent a patch Δ by a list of (offset, value) pairs. The constraints on the domain of Δ are formalized using predicates `InDirEntAlloc` and `InFat`, which test that an offset refers to the allocation state byte of a particular directory entry or lies within the FAT respectively:

```
Definition goodPatch
 (dirEnt : Z) (img : Map) (Δ : list (Z * Z)) : Prop :=
  forall (offset : Z) (value : Z),
   In (offset, value) Δ ->
   (InDirEntAlloc (offset, dirEnt, img) \/
    InFat (offset, img)).
```

Our primary lemma in this development abstracts from the heuristics discussed above by assuming that a recovered cluster chain is given. We restrict attention to chains that: have no cycles; start with the cluster listed in a given directory entry; and stay within a range of values determined by parameters in the boot block of the file system image. The nonstandard specification `fileChainDeleted` requires that a given directory entry is unallocated and a given cluster chain satisfies these properties. The standard specification `fileChain` requires that a given directory entry is allocated and a given cluster chain is found in the file system image. We have then proved that if the nonstandard specification applies to a directory entry and cluster chain, then a patch satisfying `goodPatch` exists for the standard specification on the *patched* file system image:

```
Lemma fat_deleted_yields_good_patch :
 forall (dirEnt : Z) (clusters : list Z) (img : Map),
  fileChainDeleted dirEnt clusters img ->
  (exists (Δ : list (Z * Z)),
    fileChain dirEnt clusters (img ⊕ Δ) /\
    goodPatch dirEnt img Δ).
```

[2]FAT file systems may use 12, 16, or 32 bits for an entry in the File Allocation Table; the semantics of some metadata depends on the size of entries.

Within the proof, `goodPatch` is essential to show that information required to parse the directory entry has not been altered by the patch. Similarly, this property is crucial when applying the lemma to the two heuristics described above. File content in a file system image with a deleted file is *almost* unchanged after a patch satisfying `goodPatch` is applied: the only possible change occurs when a file's content includes its own directory entry; and this pathological case is therefore excluded by the definition of fileExistsDeleted for each heuristic. This means that file content determined by a cluster chain in a file system image agrees with file content in a patched image *if* the patch writes the given cluster chain into the file system image.

VII. Related Work

In this section we describe related work in digital forensics and theorem proving.

A. Declarative Models of Forensics

Stalland and Levitt describe a declarative encoding of data invariants and a decision tree format to codify the deductions performed by a forensic examiner [11]. Their prototype tool uses an expert system to search for possible deductions, whereas we use a theorem prover to verify deductions. Similarly, Kahvedžić and Kechadi present an ontology for digital forensic knowledge [12], and provide semantics via inference rules written in the Semantic Web Rule Language [13].

Forensic procedures have been also been analyzed via formal models. [14] discusses the effectiveness of forensic procedures against attackers in the context of a simple formal model. In [15], Carrier and Spafford describe digital forensic methodologies in terms of an abstract model of a system and its history, but they do not consider the specification of concrete models.

Van den Bos and van der Storm pioneer the tool-independent specification of file formats for creation of file recognizers for forensics purposes [16].

B. Validation of Computer Forensics Software and Results

Guo et al develop and classify detailed requirements for forensic tool testing [17]. Lyle describes undesirable and inconsistent behavior of forensics tools [18], and this has led to a subsequent program of testing [19]. The Scientific Working Group on Digital Evidence [2] identify soundness problems in the output *and* interpretation of forensics tools, as well as completeness issues discussed in section III.

C. Proof Generation

Proof by Reflection: In section IV, we described the construction of proofs in forensic certificates via proof by reflection. This technique has been used in computationally-intensive formalizations, e.g., [20]. We have not yet investigated the use of architectures developed to assist with reflection [21], [22]. In [23], Chaieb and Nipkow argue that decision procedures used in proof by reflection should be abstract enough to share between theorem provers; the goal of reducing dependency on a particular theorem prover is relevant to forensic certificates.

Runtime Performance: To improve the performance of proof checking within Coq, Grégoire and Leroy show how to perform strong reduction (reduction under λ-abstractions) on a variant of the ZAM abstract machine (for the Objective Caml bytecode interpreter) using symbolic weak reduction and a readback scheme [24]. More recent work on untyped Normalization by Evaluation [25] compiles Coq programs to Objective Caml programs [26], [27].

Our formalization described in section V uses the implementation of binary integers in the Coq standard library. The extension of Coq with machine integers and persistent arrays [28] has potential to improve the runtime performance of our formalization.

Richer Programming Models: Coq's type theory is strongly normalizing and stateless, and so it can be awkward to encode some computational procedures. Several recent works [29], [30] have shown how computation in richer programming models (including, e.g., non-termination and state) can be reflected into Coq.

D. Verified SAT Solvers and Certificates

The forensic certificate architecture closely resembles that of SAT solver certificates. A SAT solver performs a computationally-intensive search for solutions, and produces a certificate for either a solution or a proof that the formula is unsatisfiable [31]. The certificate is verified by independent software. The code for the verifier, the TCB, is smaller and simpler than the code of the solver.

Two approaches have been used to integrate SAT solving with the Coq theorem prover. The first defines a SAT solver in Coq and establishes its correctness [32], [33], [34], i.e., proof by reflection is used in the search. The second uses external SAT solvers for search, and then verifies the witnesses within Coq [31], [35]. The verification is conducted by certified code, i.e., proof by reflection is used in the verification. The latter approach avoids the need to certify the complex optimizations used in a SAT solver and the limitations of the runtime systems in existing theorem provers. Our forensic certificate architecture follows the latter approach.

One glaring dissimilarity between SAT solver certificates and our forensic certificate architecture lies in the treatment of negation. For SAT solver certificates, unsatisfiability is witnessed by a resolution proof. For forensic certificates, as discussed in subsection III-C, the absence of occurrences of a property in a large interval of an image may be dealt with using external verifiers in conjunction with deduction using traditional theorem provers. The external verifiers may be, e.g., extracted from certified implementations.

VIII. Conclusion and Further Work

There are severe consequences for errors in digital forensics. In a recent trial, the defendant's computer had a record of at least one visit to a web page about chloroform [4]. Forensic examiners found that one forensic tool reported 1 visit but a second tool reported 84 visits. After the trial testimony, the cause of the discrepancy was identified by the tool developers [36], [37]: the web brower's database format only stores a record of the number of visits when the number is strictly greater than 1; and the tool reporting 84 visits read the number of visits for the next URL in the database. The bug was not evident, despite manual review of the data, because the database format in question is complex and poorly specified.

The end goal of this research is trustworthy digital forensics. Practitioners and researchers have identified the opacity of software tools as a significant impediment to making digital forensics trustworthy. Ongoing work in the community to address this situation has borrowed two important ideas from classical software engineering: (a) the clarification of software requirements for digital forensics tools, so as to identify what they are really doing, and (b) the facilitation of composition between tools to enable the construction of complex software artifacts from simpler pieces.

Our work borrows yet another idea from software construction: for software that is difficult to verify directly, one should focus instead on the correctness of individual executions. This has led us to design an architecture for forensic certificates that can be validated independently of the tools that created them in the first place. Our formal treatment of forensic certificates aims to:

- eliminate ambiguity in the forensic claims that are made;
- prevent erroneous deductions when composing the results of multiple forensic tools;
- avoid reliance on unspecified black box tools;
- make explicit reliance on unverified tools.

In this paper, we have explored the challenges in constructing such an architecture, specifically in the domain of file system forensics. In the process, we have been forced to clarify the semantics of file system specifications and the trust relationships that are often left implicit in presenting the results of forensics analysis. Our case study from a forensics challenge provides some evidence for the viability of our approach.

We have introduced a novel approach for the specification and analysis of forensic data-recovery heuristics based on providing patches that restore data structures. We have discussed this in the context of recovering deleted files. More generally, forensic certificates containing patches (with semantic constraints) can explain forensics results that do not have commonplace operations. As two examples:

- A forensics search tool may find a partially-overwritten file in a disk image and provide a patch that fills in the missing data. The missing data might be taken from an older copy of the file found in a backup. The patch can describe precisely how much of the file is present on the original disk and how much is reconstructed from the backup.
- A simple anti-forensics tool that changes the size of clusters in the metadata can make a file system appear corrupt. A forensics search tool can create a patch that undoes the alteration and justifies that the change is consistent with other information in the file system. In other cases, tools may show that no such patch is possible.

We intend this paper as part of an ongoing research program to encompass other areas of digital forensics such as timeline analysis and memory forensics.

Acknowledgements We gratefully acknowledge the comments and suggestions of the anonymous referees.

REFERENCES

[1] E. Casey, "Editorial - cutting the Gordian knot: Defining requirements for trustworthy tools," *Digital Investigation*, vol. 8, no. 3–4, pp. 145 – 146, 2012.

[2] Scientific Working Group on Digital Evidence, "Error mitigation report," https://www.swgde.org/documents/Released%20For%20Public%20Comment, 2013.

[3] E. Casey, "Tool review - WinHex," *Digital Investigation*, vol. 1, no. 2, pp. 114–128, 2004.

[4] L. Alvarez, "Software designer reports error in Anthony trial," The New York Times. http://www.nytimes.com/2011/07/19/us/19casey.html?_r=2&hp, July 2011.

[5] Honeynet Project, "Scan of the month 15," http://old.honeynet.org/scans/scan15/, 2001.

[6] S. L. Garfinkel, "Digital forensics XML and the DFXML toolset," *Digital Investigation*, 2012.

[7] B. Carrier, *File System Forensic Analysis*. Addison Wesley, 2005.

[8] J. Harrison, "Metatheory and reflection in theorem proving: A survey and critique," 1995.

[9] S. L. Garfinkel, "Anti-forensics: Techniques, detection and countermeasures," in *The 2nd International Conference on i-Warfare and Security (ICIW), Naval Postgraduate School, Monterey, CA*, 2007.

[10] Microsoft Corporation, "Microsoft extensible firmware initiative FAT32 file system specification, FAT: General overview of on-disk format," Microsoft Corporation, Tech. Rep., 2000, version 1.03.

[11] T. Stallard and K. Levitt, "Automated analysis for digital forensic science: Semantic integrity checking," in *ACSAC*. IEEE Computer Society, 2003.

[12] D. Kahvedžić and T. Kechadi, "Dialog: A framework for modeling, analysis and reuse of digital forensic knowledge," *Digital Investigation*, vol. 6, Supplement, no. 0, pp. S23 – S33, 2009, the Proceedings of the Ninth Annual {DFRWS} Conference. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S174228760900036X

[13] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "Swrl: A semantic web rule language combining owl and ruleml," World Wide Web Consortium, W3C Member Submission, 2004. [Online]. Available: http://www.w3.org/Submission/SWRL

[14] R. Leigland, "A formalization of digital forensics," *International Journal of Digital Evidence*, vol. 3, no. 2, pp. 114–128, 2004.

[15] B. D. Carrier and E. H. Spafford, "Categories of digital investigation analysis techniques based on the computer history model," *Digital Investigation*, vol. 3, pp. 121–130, 2006.

[16] J. Van den Bos and T. Van der Storm, "Bringing domain-specific languages to digital forensics," in *ICSE*, 2011, pp. 671–680.

[17] Y. Guo, J. Slay, and J. Beckett, "Validation and verification of computer forensic software tools — searching function," *Digital Investigation*, vol. 6, pp. s12–s22, 2009.

[18] J. Lyle, "Quirks uncovered while testing forensic tool," http://www.cftt.nist.gov/presentations/ENFSC-Lyle-Oct-08.ppt, 2008.

[19] ——, "Forensic tool testing results," http://www.cftt.nist.gov/presentations/NeFX-10-lyle-CFTT-test-strategy.pdf, 2010.

[20] G. Gonthier, "The four colour theorem: Engineering of a formal proof," in *ASCM*, ser. LNCS, D. Kapur, Ed., vol. 5081. Springer, 2007, p. 333.

[21] G. Gonthier and A. Mahboubi, "An introduction to small scale reflection in Coq," *Journal of Formalized Reasoning*, vol. 3, no. 2, pp. 95–152, 2010.

[22] G. Malecha, A. Chlipala, T. Braibant, P. Hulin, and E. Z. Yang, "Mirrorshard: Proof by computational reflection with verified hints," *CoRR*, vol. abs/1305.6543, 2013.

[23] A. Chaieb and T. Nipkow, "Proof synthesis and reflection for linear arithmetic," *J. Autom. Reason.*, vol. 41, no. 1, pp. 33–59, Jul. 2008.

[24] B. Grégoire and X. Leroy, "A compiled implementation of strong reduction," in *ICFP*. ACM, 2002, pp. 235–246.

[25] U. Berger, M. Eberl, and H. Schwichtenberg, "Normalization by evaluation," in *Prospects for Hardware Foundations*, ser. LNCS, B. Möller and J. Tucker, Eds. Springer Berlin Heidelberg, 1998, vol. 1546, pp. 117–137.

[26] M. Boespflug, "Conversion by evaluation," in *PADL*, ser. LNCS, M. Carro and R. Peña, Eds., vol. 5937. Springer, 2010, pp. 58–72.

[27] M. Boespflug, M. Dénès, and B. Grégoire, "Full reduction at full throttle," in *CPP*, ser. LNCS, J.-P. Jouannaud and Z. Shao, Eds., vol. 7086. Springer, 2011, pp. 362–377.

[28] M. Armand, B. Grégoire, A. Spiwack, and L. Théry, "Extending Coq with imperative features and its application to SAT verification," in *ITP*, ser. LNCS, M. Kaufmann and L. C. Paulson, Eds., vol. 6172. Springer, 2010, pp. 83–98.

[29] G. Claret, L. D. C. González-Huesca, Y. Régis-Gianas, and B. Ziliani, "Lightweight proof by reflection using a posteriori simulation of effectful computation," in *ITP*, ser. LNCS, vol. 7998. Springer, 2013, pp. 67–83.

[30] B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis, "Mtac: a monad for typed tactic programming in Coq," in *ICFP*, G. Morrisett and T. Uustalu, Eds. ACM, 2013, pp. 87–100.

[31] A. Darbari, B. Fischer, and J. Marques-Silva, "Industrial-strength certified SAT solving through verified SAT proof checking," in *ICTAC*, ser. LNCS, A. Cavalcanti, D. Déharbe, M.-C. Gaudel, and J. Woodcock, Eds., vol. 6255. Springer, 2010, pp. 260–274.

[32] S. Lescuyer and S. Conchon, "A reflexive formalization of a SAT solver in Coq," in *Emerging Trends of the 21st International Conference on Theorem Proving in Higher Order Logics*, 2008.

[33] ——, "Improving Coq propositional reasoning using a lazy CNF conversion scheme," in *Frontiers of Combining Systems, 7th International Symposium, Proceedings*, ser. LNCS, S. Ghilardi and R. Sebastiani, Eds., vol. 5749. Trento, Italy: Springer, Sep. 2009, pp. 287–303.

[34] S. Lescuyer, "Formalizing and implementing a reflexive tactic for automated deduction in coq," Ph.D. dissertation, Université Paris-Sud, 2011.

[35] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner, "A modular integration of SAT/SMT solvers to Coq through proof witnesses," in *CPP*, ser. LNCS, J.-P. Jouannaud and Z. Shao, Eds., vol. 7086. Springer, 2011, pp. 135–150.

[36] Digital Detective, "About digital evidence discrepancies – Casey Anthony trial," http://wordpress.bladeforensics.com/?p=357, 2011.

[37] CacheBack, "Computer evidence in the Casey Anthony trial - a post mortem," http://pdfserver.amlaw.com/ltn/Computer_Evidence_in_the_Casey_Anthony_Trial_a_Post_Mortem.pdf, July 2011.

[38] J.-P. Jouannaud and Z. Shao, Eds., *Certified Programs and Proofs (CPP)*, ser. LNCS, vol. 7086. Springer, 2011.