Hindawi
Security and Communication Networks
Volume 2022, Article ID 1322264, 17 pages
https://doi.org/10.1155/2022/1322264

Research Article

Proving Reliability of Image Processing Techniques in Digital Forensics Applications

Saima Iqbal,[1] Wilayat Khan,[1] Abdulrahman Alothaim,[2] Aamir Qamar,[1] Adi Alhudhaif,[3] and Shtwai Alsubai[3]

[1]*Department of Electrical and Computer Engineering, COMSATS University Islamabad, Wah Campus, Islamabad, Pakistan*
[2]*Department of Information Systems, College of Computer and Information Sciences, King Saud University, Riyadh 11451, Saudi Arabia*
[3]*Department of Computer Science, College of Computer Engineering and Sciences in Al-Kharj, Prince Sattam Bin Abdulaziz University, Al-Kharj, Saudi Arabia*

Correspondence should be addressed to Wilayat Khan; wilayat@ciitwah.edu.pk

Received 17 November 2021; Revised 28 January 2022; Accepted 5 February 2022; Published 31 March 2022

Academic Editor: Farhan Ullah

Binary images have found its place in many applications, such as digital forensics involving legal documents, authentication of images, digital books, contracts, and text recognition. Modern digital forensics applications involve binary image processing as part of data hiding techniques for ownership protection, copyright control, and authentication of digital media. Whether in image forensics, health, or other fields, such transformations are often implemented in high-level languages without formal foundations. The lack of formal foundation questions the reliability of the image processing techniques and hence the forensic results loose their legal significance. Furthermore, counter-forensics can impede or mislead the forensic analysis of the digital images. To ensure that any image transformation meet high standards of safety and reliability, more rigorous methods should be applied to image processing applications. To verify the reliability of these transformations, we propose to use formal methods based on theorem proving that can fulfil high standards of safety. To formally investigate binary image processing, in this paper, a reversible formal model of the binary images is defined in the Proof Assistant Coq. Multiple image transformation methods are formalized and their reliability properties are proved. To analyse real-life RGB images, a prototype translator is developed that reads RGB images and translate them to Coq definitions. As the formal definitions and proof scripts can be validated automatically by the computer, this raises the reliability and legal significance of the image forensic applications.

1. Introduction

Image processing in general is widely used in the computer-based tools and techniques in different applications including digital forensics, health, crack detection in concrete, obstacle detection, and astronomy. The modern quality digital cameras and the sophisticated photo editing tools have made image-based applications very attractive for people, in particular, in social media and mobile applications. However, the same powerful photo editing software have made digital image fabrication and forgery very easy [1]. Such forgeries appear authentic to human eye but are unacceptable to the law enforcement agencies [2]. Image processing techniques are used for identifying fabricated documents using digital forensics analysis [3]. The reliability of the processes implemented by the digital forensics practitioners, however, must be ensured [4]. In a typical digital forensics application, the raw binary data is normally derived as a binary image. During the forensic examination of a hard drive, for example, a binary image of the entire hard drive is created and analysed. Image forensic has been used in forensic document and algorithmic handwriting analysis [5]. The applications of the binary and grayscale images are so versatile that they create the need of their forensic analysis. The digital forensics analysis is used to find malicious manipulations in the image [6]. In fact, digital

image forensic has emerged as a new research field aimed at validating the origin authenticity of the images [7]. Nevertheless, the counter-forensics has also risen up to impede or mislead forensic analysis of the digital images [8].

In digital forensics applications or otherwise, binary images are often quantized to two extreme tones/intensity values, where black (0 intensity value) is usually represented with a 1, whilst white (255 intensity value) is represented as 0. Binary images are usually created by an information abstraction process from a gray channel after applying either thresholding or any other segmentation method. A standard RGB image is comprised of three gray channels, packed in a predefined sequence of the red, green, and blue colors [9]. A binary conversion leads to a significant loss of information; however, if this conversion process is handled with care and an appropriate segmentation algorithm is applied, salient/key information can be preserved. Formally, the binary operations are carried out using a sliding window of variable sizes (e.g., 3×3 or 5×5). Conventionally, a window center plays a pivotal role and the standard operations including median, mean, mode, and so on, are carried out to update the center location of the sliding window. Amongst several, a few operations on a binary image are run-length encoding, dilation, erosion, close, open-close, close-open, and skeletonization to name but a few.

The RGB and gray-scale images have more internal details, but they require more storage and bandwidth to store and transmit. Furthermore, applications based on the color or gray-scale images are computationally extensive and require expensive equipment and computer capabilities to process such images [10]. Binary images, on the other hand, have only two gray values, 1 (black) or 0 (white), and require only one bit per pixel storage (regardless of the bit assignment to the black and white colors, the formal definitions and proofs are still valid). Binary images are created in a number of ways: they may be delivered directly by sensors or indirectly, and more often, they are created from the gray-scale images. Among the major objectives of using the binary images are simplified processing, transmission, storage, and printing.

To achieve high standards of security and reliability of applications based on image transformations, they need to be formally verified. To enable formal reasoning, such transformations need to be specified and reasoned about using mathematical tools and techniques. The conventional methods based on simulation and testing can be used for systems verification; however, these methods can only show presence of faults but cannot show their absence. There are tools based on model checking, but they might face state or memory explosion problems. Formal verification methods based on interactive proof assistants, such as Coq and Isabelle/HOL, are more powerful and expressive and do not face state explosion issues. A proof assistant is a computer-based tool that is guided by a proof engineer in the step-by-step proof process. The interactively generated proof script can be checked by the computer.

To the best of our knowledge, there is no mechanized proof of properties of the binary image processing algorithms described in this paper. To formally prove the correctness of the transformations carried over the binary images, the images and the transformations must be defined in a formal language with a proof facility. In this paper, we are using Coq for the formal specification and verification of the binary image processing. In this approach, as depicted in Figure 1, an RGB image is first read and then converted to a binary image. The binary image is translated to formal representation (model) of binary images as defined using the Coq notations. The Coq definition of the binary image is reversible: a binary image matrix can be generated back from the Coq definition of the binary image. In addition to the binary image, a number of transformations on the binary images and their properties are also formally defined. Using the Coq proof commands (tactics), interactive proofs of properties are carried out and then checked using the Coq proof checker facility. At the end, the binary image defined in the Coq is translated back to the binary image matrix of 0's and 1's (see Section 6).

The following are the major contributions of this paper.

(i) Light-weight formal models of the gray-scale and binary images are built in the Coq tool (described in Section 4.1).

(ii) Several image transformations and an image compression technique are formally defined (discussed in Sections 4.2–4.5).

(iii) A number of properties of the binary images and operations over them, such as involution, area size, and distance measures, are stated and proved in Section 5.

(iv) A prototype translator is developed to translate an RGB image to a binary image and generate other Coq definitions. The translator automatically generates a proof that the translation to binary image and other definitions is sound. Furthermore, the translator converts the Coq binary image back to a binary image matrix. The translator and their experimental analysis are given in Section 6.

The formal developments in this paper formalize binary image operations and carry proof of soundness of these operations. The formal proofs of the binary image operations guarantee the soundness of any application based on these operations, including but not limited to forensics, health, and engineering. Complete Coq script including formal definitions of the binary/grayscale images, image transformations, proof of soundness of image transformations, and source code of the translator is available at our GitHub repository at https://github.com/wilstef/binaryimagescoq. The rest of the paper is organized as follows. A review of the research contributions in the binary image processing, formal hardware, and software verification is included in the next section. In Section 3, the Coq Proof Assistant and the binary image processing are introduced. In Section 4, the gray-scale and binary images are formally specified and a number of operations on them are defined. Several properties of the image processing and encoding are defined and proved in Section 5. An experimental analysis of the translator and formal definitions is carried out in Section 6. The paper is concluded in Section 7.

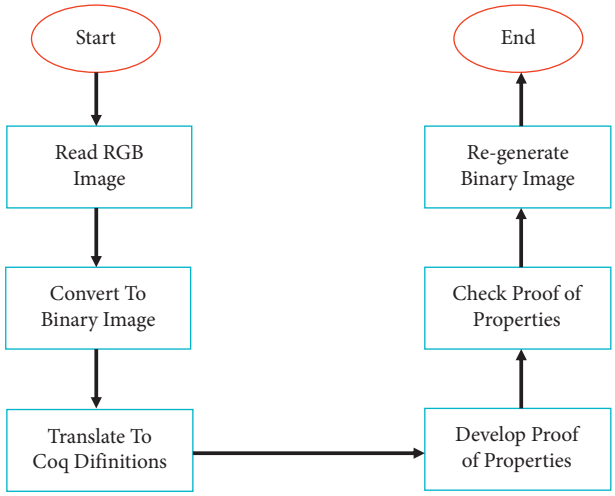


Figure 1: Formal image processing verification approach.

2. Literature Review

In this section, a summary of the research contributions in the binary image processing and formal hardware and software systems' verification is given.

2.1. Binary Image Processing. Binary image processing has found its place in a number of applications including but not limited to agriculture and engineering. Hernández et al. [10] used binary image analysis in their open-source methodology to measure water elevations in two-dimensional hydrodynamic experiments. To detect winding deformation faults in power transformers, frequency response analysis has been widely used as a diagnostic tool. Zhao et al. [11] introduced fault detection method based on the analysis of binary images obtained from frequency response analysis signatures. To detect cracks in concrete using unmanned aerial vehicles, Kim et al. [12] used binary image analysis to transform the crack and background into binary images. They identified the crack objects by categorizing the pixels as black whose gray values are less than a threshold value.

Formal verification has been applied in a number of domains and applications, including image processing tools. Narendran and Stillman formally verified the description of an image processing chip Sobel [13] using theorem proving approach. During their verification attempt, they found several design errors in the circuit description of the design under test. Bourbakis and Alexopoulos [14] defined a special-purpose context-free grammar for the language SCAN. The language SCAN was devoted to describing and generating a range of two-dimensional array accessing algorithms used for image processing. They defined a formal definition of SCAN and described the underlying method for spatial access.

Ehlers [15] presented an approach for formally verifying piecewise linear feed-forward neural networks. Their formally verified approach was evaluated on critical applications, such as obstacle detection and handwritten digit recognition. To protect against adversarial attacks on convolutional neural networks (CNNs), Kouvaros and Lomuscio [16] formally verified CNN-based perception systems. The authors formally verified the robustness of the local image transformations. Sun et al. [17] formally verified the safety of autonomous robot controlled by a neural network. The neural network processes images to produce control actions for avoiding obstacles. None of these researchers have applied proof assistants in the formal verification of these systems.

2.2. Formal Hardware Verification. Formal verification techniques have been applied to study computer hardware systems. Meredith et al. [18] defined an executable semantics by embedding hardware description language (HDL) Verilog in the theorem prover [19]. Inspired from [18], Khan et al. [20, 21] defined an HDL, dubbed as VeriFormal, in the more powerful and expressive higher-order logic of proof assistant Isabelle/HOL. VeriFormal is available with an executable simulator and a prototype translator to translate existing design descriptions in Verilog to VeriFormal. The HDL VeriFormal can be used to design hardware circuits, simulate them, and formally verify their properties using Isabelle/HOL. Later on, the HDL VeriFormal and its simulator were used to formally prove functional equivalence of several logic circuits [22].

Braibant and Chlipala [23] defined a simplified version of the high-level language Bluespec, called Fe-Si. Their language Fe-Si has been deeply embedded in the Proof Assistant Coq. In a similar contribution, Choi et al. [24] developed a formalized version of Bluespec in Coq, called Kami. The objective of platform Kami was to develop high-level parametric hardware specifications. Both, Fe-Si [23] and that of Choi et al. [24], are different from the HDL VeriFormal in the level of hardware specification. The high-level language Bluespec is used for high-level hardware specification, while the HDL VeriFormal is a low-level language at the register-transfer logic level.

2.3. Formal Software Verification. Software systems, in particular those used in business or life critical systems, must be ensured to behave as desired using formal tools. Bugliesi et al. used formal techniques to study web session integrity [25, 26] and web session confidentiality [27, 28]. In addition to the formal analysis, they also developed prototype browser extension as the proof of concept. Khan et al. defined a formal model, dubbed as CrashSafe, of inter-component communication within Android systems in Coq proof assistant. The authors formally showed that their model can be used to capture faults in the Android applications. They later on formally analysed Android systems' security using language-based security techniques [22].

Alturki et al. [29] developed a state-transition formal model of the Algorand consensus protocol and verified its asynchronous safety using Coq proof assistant. The authors claim that their model is general and can be adopted to prove other properties (e.g., liveness) of the protocol. Anand et al. [30] developed and mechanically verified an optimized compiler, called CertiCoq, for Coq programs in Coq proof

assistant. Leroy et al. [31] developed a formally verified compiler, CompCert, for the C programming language. The main advantage of this compiler is that the executable code produced by CompCert is proved to behave as specified by the C semantics.

3. Background

The formal definitions and proofs of binary image processing are carried out in the Proof Assistant Coq. In this section, the basics of Coq and binary image processing are introduced.

3.1. Coq Proof Assistant. To formally verify systems' properties, the system and the properties of interest should be specified in the logic of a proof assistant (e.g., Coq [32, 33] and Isabelle/HOL [34]). The proof assistant can be used to build a proof that the (model of the) system satisfy the properties. The proof checker facility of the proof assistant is then used to mechanically check if the proof is valid. To describe the formal developments and proofs using a proof assistant, a simple system of natural numbers is defined and reasoned about using the proof assistant Coq. To begin with, numbers are inductively defined as data type nat using the keyword Inductive with two constructors for generating elements of the type nat (lines 1–3, Listing 1). The definition of nat states that O (for 0) is nat and if n is nat, then S n is also nat. The term S (S (S (S O))), for example, is a natural number 4 in nat.

Next, we define a recursive function add (lines 5–9) on the numbers just defined. The function returns the second argument m if the first argument is O and it returns S (add n′ m) if the first argument is of the form S n′. A lemma add_n_o, that is add n O = nm holds for any value of n is stated and proved in Listing 1 (lines 11–18). A proof begins with the proof command Proof and ends with Qed. Each proof command, also called tactic, ends with a dot (.), and multiple tactics can be combined into a sequence using semicolon (;). Comments can be introduced anywhere within the script using the syntax (∗ a comment ∗). The lemma add_n_o is proved using induction on the construction of the first argument n. During the proof process, the proof assistant is guided interactively by providing tactics (lines 12–18). The correctness of the proof script just created (lines 12–18 in Listing 1) can be mechanically checked using the Coq proof checker program.

3.2. Binary Images. Binary images consist of picture elements (pixels) with two gray values black and white, denoted as 1 and 0, respectively. The color depth of the binary images is just one bit per pixel. It plays a significant role in the reduced size of such images. Binary image representation is used in a multitude of image applications, including but not limited to digital forensics, optical character recognition, edge detection, optical measurement applications, mathematical morphology, and object tracking and orientations.

The upper binary image in Figure 2 represents the letter N. The resolution of (number of pixels represented) this image is 16 and the depth is one bit per pixel. In the lower image of Figure 2, each pixel is designated with either 1 or 0 depending on its color (black is 1 and white is 0). The bits matrix on the right side of the figure represents the binary image in bits.

4. Formalization of Binary Images

To reason about binary image processing, the concepts of the pixels, binary images, and grayscale images are first defined in the Coq Proof Assistant. In a binary image, every pixel has only two values or colors: black and white. On the contrary, in grayscale images, the pixels' color value ranges over 0–255 different values.

4.1. Binary and Gray-Scale Images. To begin with, first, the data type color is inductively defined in the code Listing 2. The data type color has two possible values or constructors white and black.

A pixel of the binary image is defined as a type pixel as shown in Listing 3. The type pixel is defined as a tuple of the row, column, and a color. It has one constructor pixel that takes three arguments of type nat and color. The first two arguments of the type nat define the row and column position while the third argument defines the color of a pixel. On line 4, the shorthand notation B{r,c,col} is defined to represent a pixel at coordinates (r,c) with color intensity col. Finally, a binary image is defined as a list of binary pixels (Listing 4). To facilitate the reuse of the formal definitions, Coq allows notations. As described in the introduction, the black and white colors are assigned digits 1 and 0, respectively (Listing 5).

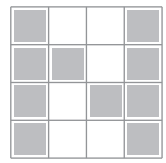
To highlight the application of the formal definitions in Listings 2–4, the binary image in Figure 2 is encoded using the formal definitions developed (lines 1–4, Listing 6). Using the notations for the colors black and white from Listing 5, the same image is encoded more compactly (lines 6–9, Listing 6).

Similarly, a grayscale pixel is defined as a gspixel type in Listing 7. It has a single constructor GSPixel with three arguments of type nat for row, column, and grayscale color of the pixel. The grayscale image gsimage is defined as a list of grayscale pixels (line 4, Listing 7). The notations for binary and grayscale pixels are defined as B{row,column, color} and G{row,column,gs-color}, respectively, where row, column and gs-color are natural numbers, while color is either white or black value (Listings 8 and 9).

4.2. Basic Operations on the Pixels and Images. To state and prove interesting properties of binary and grayscale images, different operations over pixels and images are first defined in Listing 10. These operations include equality of colors eqbcol (lines 1–6), negation of a color negcolor (lines 8–12), negation of a pixel color negpix (lines 14–17), negation of a binary image negimage (lines 19–23), equality of pixels eqpixel (lines 25–29), and negation of a specific pixel in an image negpiximg (lines 31–37), respectively. The first function eqbcol defines when two colors are equal. It returns

```
Inductive nat: Type:=
  |O: nat
  |S: nat ⟶ nat.
Fixpoint add (n m: nat): nat:=
  match n with
  |O ⇒ m
  |S n′ ⇒ S (add n′ m)
  end.
Lemma add_n_o: ∀ n, add n O = n.
Proof.
  induction n.
+ (* CASE 1: n is O *)
reflexivity.
  + (* CASE 2: n is (S n) *)
  simpl. rewrite IHn. auto.
Qed.
```

Listing 1: Interactive formal proof in Coq.



1	0	0	1	1	0	0	1
1	1	0	1	1	1	0	1
1	0	1	1	1	0	1	1
1	0	0	1	1	0	0	1

Figure 2: Binary image of the letter N.

```
Inductive color: Type:=
  |white: color
  |black: color.
```

Listing 2: Data type color.

```
Inductive pixel: Type :=
 |Pixel (r c: nat) (col: color).
Notation "B r,c, col" (Pixel r c col).
```

Listing 3: Data type pixel.

```
Definition image: = list pixel.
```

Listing 4: Definition of the binary image.

```
Notation "1": = (black).
Notation "0": = (white).
```

Listing 5: Shorthand notations representing black and white colors.

```
[B{0, 0, black}; B{0, 1, white}; B{0, 2, white; B0, 3, black};
B{1, 0, black}; B{1, 1, black}; B{1, 2, white}; B{1, 3, black}; B{2, 0, black}; B{2, 1, white}; B{2, 2,
black}; B{2, 3, black};
B{3, 0, black}; B{3, 1, white}; B{3, 2, white}; B{3, 3, black}].
[B{0, 0, 1}; B{0, 1, 0}; B{0, 2, 0}; B{0, 3, 1};
B{1, 0, 1}; B{1, 1, 1}; B{1, 2, 0}; B{1, 3, 1};
B{2, 0, 1}; B{2, 1, 0}; B{2, 2, 1}; B{2, 3, 1};
B{3, 0, 1}; B{3, 1,0}; B{3, 2, 0}; B{3, 3, 1}].
```

Listing 6: Encoding binary image of Figure 2.

```
Inductive gspixel: Type :=
  |GSPixel (r c v: nat).
Definition gsimage: = list gspixel.
```

Listing 7: Definition of grayscale pixel and image.

```
Notation "B r, c, col": = (Pixel r c col).
Notation "G r, c, v": = (GSPixel r c v).
```

Listing 8: Shorthand notations representing the binary and gray-scale images.

true (of type bool) if both the colors are either white or black; otherwise, it returns false.

Given one color, the function negcolor (lines 8–12) returns the other color. It uses pattern matching (match . . . with) that looks for the two patterns, black and white, of the color. The function negpix just inverts the color of the given pixel while keeping the row and column unchanged. Using negpix, the function negimage inverts the color of each pixel in the image. The next function eqpixel defines equality of two pixels. Two pixels are equal if their rows, columns, and colors are the same. This function is used in the function negpiximg, which inverts a specific pixel in an image.

4.3. Thresholding. In the binary image processing, thresholding is a technique used for converting a gray-scale image to a binary image. It is used to separate subimages that represent objects from the background. If a camera is designed to give gray-scale images, then thresholding is used to create binary image from the gray-scale image. A binary image B is retrieved by applying a threshold transformation T to a gray-scale image G. Informally, a fixed type of thresholding operation is defined (in equations (1) and (2)) below.

$$B[i, j] = G_T[i, j], \tag{1}$$

where

$$G_T[i, j] = \begin{cases} 1, & \text{if } G[i, j] \le T, \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

This kind of thresholding operation over a gray-scale image is defined in the Coq as a recursive function thresholding as shown in Listing 11. It takes an image and a fixed threshold value (of type nat) and returns threshold binary image. This function replaces the pixel gray-scale color value with 1 (black) if the gray-scale value of the pixel is less than or equal to the threshold provided; otherwise, it turns the color of the pixel to 0 (white).

In some applications, if the intensity values of the object of interest are in a range, then a range of thresholding values T_1 and T_2 is used. In this thresholding technique, the gray-scale pixel is turned black (represented with 1) if the gray-scale value of a pixel is in the range of T_1 and T_2. In other

```
Definition eqbcol (c1 c2: color): bool :=
match c1, c2 with
  |white, white ⇒ true
  |black, black ⇒ true
  |_, _ ⇒ false
end.
Definition negcolor (c: color): color:=
match c with
  |white ⇒ black
  |black ⇒ white
end.
Definition negpix (p: pixel): pixel:=
match p with
  |Br,c,col ⇒ Br,c, negcolor col
end.
Fixpoint negimage (pic: image): image:=
match pic with
  |nil ⇒ nil
  |cons p tl ⇒ cons (negpix p) (negimage tl)
end.
Fixpoint eqpixel (p1 p2: pixel): bool :=
match p1, p2 with
  |Br,c,col, Br′,c′,col′ ⇒
    andb (andb (r =? r′) (c =? c′)) (eqbcol col col′)
end.
Fixpoint negpiximg (p: pixel) (img: image): image:=
match img with
  |nil ⇒ nil
  |cons p′ ⇒ if eqpixel p p′ then
    cons (negpix p′) (negpiximg p tl)
    else cons p′ (negpiximg p tl)
end.
```

Listing 9: Definitions of operations over pixels and images.

```
Compute (thresholding gsimage 170).
Compute gsimagecoq.
Definition binaryimage := (thresholdrange gsimage 100 200).
Compute (areasize binaryimage).
Compute (runlength binaryimage).
Compute (sumrunlen (runlength binaryimage)).
```

Listing 10: Operations over the image.

```
Fixpoint thresholding (img: gsimage) (thresh: nat): image :=
match img with
|nil ⇒ nil
|cons Gr,c,gsv tl ⇒
  if (gsv <=? thresh)
  then cons Br,c,black (thresholding tl thresh)
  else cons Br,c,white (thresholding tl thresh)
end.
```

Listing 11: Definition of fixed thresholding.

```
Fixpoint thresholdrange (img: gsimage) (T1 T2: nat): image :=
match img with
|nil ⇒ nil
|Gr,c,gsv:tl ⇒
   if andb (T1 <=? gsv) (gsv <=? T2)
   then Br,c,black::(thresholdrange tl T1 T2).
   else Br,c,white::(thresholdrange tl T2 T2)
end.
```

Listing 12: Definition of range thresholding.

words, the binary color of the pixel is black if the gray-scale value of the corresponding pixel is less than or equal to T_2 and greater than or equal to T_1. Thresholding based on a range of threshold values is defined as follows.

$$G_T[i, j] = \begin{cases} 1, & \text{if } T_1 \leq G[i, j] \leq T_2, \\ 0, & \text{otherwise.} \end{cases} \quad (3)$$

Thresholding operation in equation (3) is defined as a function thresholdrange in Coq as shown in Listing 12. The function thresholdrange takes a gray-scale image and two threshold values and returns a threshold binary image.

4.4. Area Size. The area of all objects in a binary image is the sum of all 1's (black) intensity values. Informally, the area A of all objects in a binary image B is defined in the equation below.

$$A = \sum_{i=1}^{n} \sum_{j=1}^{m} B[i, j]. \quad (4)$$

In Coq, the area of objects is defined by the recursive function areasize in Listing 13. It takes an image and returns the total area of the objects as a natural number.

4.5. Run-Length Encoding. Run-length encoding is a compact representation of a binary image normally used for image transmission. In this encoding, the lengths of the runs of 1 (black) pixels in the image compactly represent the image. Run-length of an image may be calculated using two methods. In one method, the start position and length (repetitions) of 1's in a row are used. In the second one, the lengths of 1 and 0 runs are used; however, it must begin with 1 run. The later representation is more compact and is formalized in this section.

In the first step, run-length of the 0's (white pixels) is calculated using the recursive function runlenwhite (Listing 14). The function runlenwhite takes an image img and a list of values l and returns a list of values, where each value represents length of 0 runs. It adds the run-lengths to the provided list initialized to nil (second argument of runlenwhite). When the pixel at the head of the image is white (lines 5–6), it appends 1 at head if the list l is nil (line 5); otherwise, it increments the value at head (line 6). The run-length value h preceded with constructor S increments h by one. When the color of the pixel is black (lines 7–8), it just moves on to the next pixel with the same list l if 0 is at the head of list l (line 7); otherwise, it adds 0 at the head of the list and repeats the cycle (line 8). At the end, the order of the list of values, created for the binary image, is reversed (lines 2–3) using the function rev.

Similarly, in the second step, the run-length of the 1's (black pixels) is calculated using the recursive function runlenblack (Listing 15). It returns the list of values (of type nat), where each value represents the run-length of 1's. Next, we define a recursive function alternate (Listing 16) that appends two lists by alternatively taking one value from each list.

Finally, the run-length of a binary image is defined in the function runlength (Listing 17). This function calculates the run-length of the image with either 0 or 1 run at the head of the returned list depending on which run occurs first. Alternatively, the run-length of the image with 1 run first is defined in the function runlength′ in Listing 18.

To state interesting theorems, we need a summation function to add all the run-lengths of a binary image. This is defined in the recursive function sumrunlen in Listing 19. The function sumrunlen takes black or white run-lengths of an image (as a list of nat) and returns the sum of all runs.

5. Proof of Properties

After binary and gray-scale images are formally defined, we are now ready to state and prove different properties of the binary images and image transformations.

5.1. Involution. The involution property states that double negation of a binary image makes no change to the original image. The involution property is defined as a Lemma negimg_involution in Listing 20. This lemma is proved by induction on the argument img of type image. It additionally applies an axillary lemma negpix_involution (Listing 21) using the rewrite tactic. The negpix_involution lemma states that double negation does not change the pixel.

5.2. Run-Length and Area. Run-length encoding can be used in computing horizontal projection, the area of all the objects, and vertical and diagonal projections. In this section, we prove that the area of all the objects in an image is equal

```
Fixpoint areasize (img: image): nat:=
match img with
|nil ⇒ 0
|Br,c,col:tl ⇒ match col with
   |black ⇒ S (areasize tl)
   |white ⇒ areasize tl
   end
end.
```

Listing 13: Calculating object area.

```
Fixpoint runlenwhite (img: image) (l: list nat): list nat \coloneq
match img, l with
| nil, 0:tl″ ⇒ rev tl″
| nil, _ ⇒ rev l
| Br,c,white:tl′, nil ⇒ runlenwhite tl′ (1:nil)
| Br,c,white:tl′, h:tl″ ⇒ runlenwhite tl′ (S h:tl″)
| Br,c,black:tl′, 0:tl″ ⇒ runlenwhite tl′ l
| Br,c,black:tl′, _ ⇒ runlenwhite tl′ (0:l)
end.
```

Listing 14: Run-length of 0's.

```
Fixpoint runlenblack (img: image) (l: list nat): list nat:=
match img, l with
|nil, 0:tl″ ⇒ rev tl″
|nil, _ ⇒ rev l
|Br,c,black:tl′, nil ⇒ runlenblack tl′ (1:nil)
|Br,c,black:tl′, h:tl″ ⇒ runlenblack tl′ (S h:tl″)
|Br,c,white:tl′, 0:tl″ ⇒ runlenblack tl′ l
|Br,c,white:tl′, _⇒ runlenblack tl′ (0:l)
end.
```

Listing 15: Run-length of 1's.

```
Fixpoint alternate (l1 l2: list nat): list nat:=
match l1, l2 with
|nil, _ ⇒ l2
|_, nil ⇒ l1
|h1:tl1, h2:tl2 ⇒ h1:h2::(alternate tl1 tl2)
end.
```

Listing 16: Intermixing two lists.

to the sum of the lengths of all 1 runs. Formally, this is stated as a lemma runlenblack_eq_area_all in Listing 22.

The lemma runlenblack_eq_area_all states that the sum of the 1 (black) runs of a binary image img is equal to the area of all the objects in the image. This lemma is proved using induction on the argument img, followed by case analysis over the pixels p. In addition, the proof of this lemma includes the application of few other lemmas such as runlen_0_nil and sumrunlen_l. Complete proof script of these additional lemmas is not included here due to limited space and is available at our GitHub repository.

5.3. Distance Measures. Many applications [35, 36] require to measure the distance between any two pixels of a binary image. There are several methods to find the distance between any two pixels of the image. Among them are Euclidean, City-block, and Chessboard. The City-block

```
Definition runlength (img: image): list nat :=
match img with
|nil ⇒ nil
|Br,c,black:tl′ ⇒
   alternate (runlenblack img nil) (runlenwhite img nil)
|Br,c,white:tl′ ⇒
   alternate (runlenwhite img nil) (runlenblack img nil)
end.
```

Listing 17: Run-length of binary image.

```
Definition runlength′ (img: image): list nat :=
  alternate (runlenblack img nil) (runlenwhite img nil).
```

Listing 18: Run-length of binary image (1 run first).

```
Fixpoint sumrunlen (runl: list nat): nat :=
match runl with
| nil ⇒ 0
|r:tl ⇒ r + (sumrunlen tl)
end.
```

Listing 19: Adding the run-lengths of an image.

```
Lemma negimg_involution: forall img: image,
   negimage (negimage img) = img.
Proof.
   intros.
   induction img.
   + simpl. auto.
   + simpl. rewrite IHimg.
      rewrite negpix_involution.
      auto.
Qed.
```

Listing 20: Involution property of images.

```
Lemma negpix_involution: forall pix: pixel,
   negpix (negpix pix) = pix.
Proof.
   intro.
   destruct pix.
   simpl.
   induction col.
   + simpl. auto.
   + simpl. auto.
Qed.
```

Listing 21: Involution property of pixels.

```
Lemma runlenblack_eq_area_all: forall img:image
sumrunlen (runlenblack img nil) = areasize img.
Proof.
  intro.
  induction img as [ |p pl].
  + simpl. auto.
  + destruct p.
    destruct col.
    −simpl.
      rewrite < −IHpl.
      rewrite runlen_0_nil.
      auto.
    −simpl.
      rewrite < −IHpl.
      simpl.
      rewrite sumrunlen_l.
      simpl.
auto.
Qed.
```

Listing 22: Run-length and area.

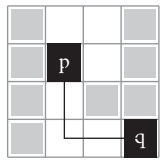


Figure 3: City-block distance measure.

distance measure between two pixels p and q is shown in Figure 3. The two pixels, p and q, are located at the positions (1,1) and (3,3), respectively. The City-block method for distance measure is informally defined as $D_{\text{city}} = |i_1 - i_2| + |j_1 - j_2|$, where (i_1, j_1) and (i_2, j_2) represent positions of the two pixels. The City-block distance between pixels p and q in the example above is $D_{\text{city\_pq}} = |1-3| + |1-3| = 2+2 = 4$.

Regardless of the distance measure method, all pixels of the image must satisfy the following properties. For any three pixels p, q, and r, we must prove the following:

(1) $d(p,q) \geq 0$ and $d(p,q) = 0$ iff $p = q$.

(2) $d(p,q) = d(q,p)$.

(3) $d(p,r) <= d(p,q) + d(q,r)$.

In this section, we formally define City-block method for computing distance between any two pixels and then state and prove the above three properties of City-block distance. To begin with, first, the operation $|m - n|$, for any m and n, is formally defined as a Coq definition modminus in Listing 23. The shorthand notation used for the function modminus is {m − −n}.

The City-block method of distance measure is defined in the function citydist in Listing 24. This function gets two pixels as argument and returns the distance between them. After defining the City-block distance measure method, we are now ready to reason about it. In the lemma distcb_min in the Listing 25, we state and prove one of the two parts of the first property. The first part of this property is $d(p,q) \geq 0$, where p and q are any two pixels of the image. This lemma is proved using case analysis on the arguments p and q.

The second part of the first property of the City-block distance measure is $d(p,q) = 0$ iff $p = q$. This property is stated as a lemma distcb_eq_0 and proved in Listing 26. The proof of this lemma proceeds using case analysis of arguments p and q using tactic destruct and then applying other Coq tactics such as rewrite, simpl, inversion, and lia.

The reverse ∀ p q, citydist p q = 0->eqpixel p q = true. of lemma distcb_eq_0 (Listing 26) is not true. The reason is that the definition of citydist does not take the color of the pixels into consideration. So even if citydist p q = 0 holds, it does not mean their colors are also equal; hence, the equality of pixels cannot be proved.

The second property states that the distance measure is commutative. That is, for any two pixels p and q, $d(p,q) = d(q,p)$. This is formalized in Coq as a lemma distcb_comm in Listing 27. Similarly, the third property, $d(p,r) <= d(p,q) + d(q,r)$, is defined as a lemma citydist_trans in Listing 28 (due to space limitations, complete proofs of these two lemmas are not listed here and can be accessed from our repository).

6. Experimental Analyses

The binary image in Figure 2 is a small 16-pixel grayscale image. To test that our formal definitions can be used for the real life images, we performed experiments by running the algorithms over RGB images. For this purpose, we designed and developed a prototype tool in C++ programming language. When run, the tool asks to enter an RGB image name and a threshold value for binary conversion. It then generates Coq proof script for the image (file proofofgsi.v), binary image matrix (text file "binaryimage.txt"), and Coq

```
Definition modminus (n m: nat): nat \coloneq
if n <? m then m – n else n – m.
Notation "m – n" \coloneq (modminus m n).
```

Listing 23: Definition and notation of modminus operation.

```
Definition citydist (p1 p2: pixel): nat \coloneq
match p1, p2 with
| Br1,c1,_, Br2,c2,_ =>  r1–r2 + c1–c2
end.
```

Listing 24: Calculating distance using the city-block method.

```
Lemma distcb_min: forall p q, citydist p q >= 0.
Proof.
intros.
destruct p as [r1 c1 col1].
destruct q as [r2 c2 col2].
simpl.
lia.
Qed.
```

Listing 25: Proof of lemma distcb_min.

```
Lemma distcb_eq_0: for all p q, eqpixel p q = true->citydist p q = 0.
Proof.
intros.
destruct p as [r1 c1 col1].
destruct q as [r2 c2 col2].
simpl.
unfold eqpixel in *.
do 2 rewrite andb_true_iff in H.
inversion H.
inversion H0.
assert (r1 = r2) as Hreq.
apply beq_nat_true; auto.
assert (c1 = c2) as Hceq.
apply beq_nat_true; auto.
rewrite Hreq. rewrite Hceq.
unfold modminus.
destruct ltb.
destruct ltb.
lia.
lia.
destruct ltb.
lia.
lia.
Qed.
```

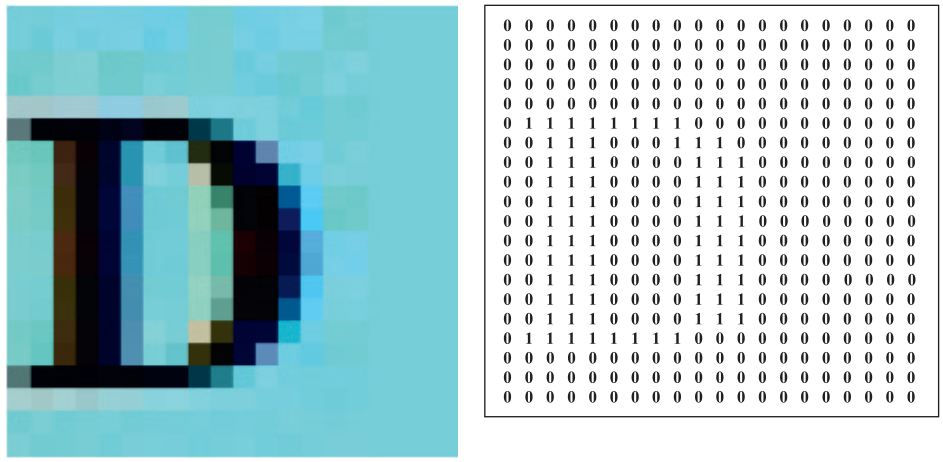
Listing 26: Proof of lemma distcb_eq_0.

```
Lemma distcb_comm: forall p q, citydist p q = citydist q p.
```

Listing 27: Proof of lemma distcb_comm.

```
Lemma citydist_trans: forall p q r
citydist p r <= citydist p q + citydist q r.
```

Listing 28: Proof of lemma citydist_trans.



(a)

(b)

```
Grayscale image matrix:
[203; 200; 201 . . . 201; 201; 201]

Grayscale image:
[G{0,0,203};G{0,1,200};G{0,2,201} . . .
G{19,16,201};G{19,17,201};G{19,18,201}].

Binary image (threshold 100):
[B{0,0,0};B{0,1,0};B{0,2,0} . . .
B{19,16,0};B{19,17,0};B{19,18,0}]
```

(c)

Figure 4: Example 1: translating the RGB image to the matrix and Coq definition gsimage. (a) RGB image. (b) Binary image matrix. (c) Coq definitions of the image.

definition of the binary image (text file "coqbinaryimage.txt"). The Coq script file includes the following:

(1) Definitions of the grayscale image matrix, grayscale image and binary image (all are tool generated)

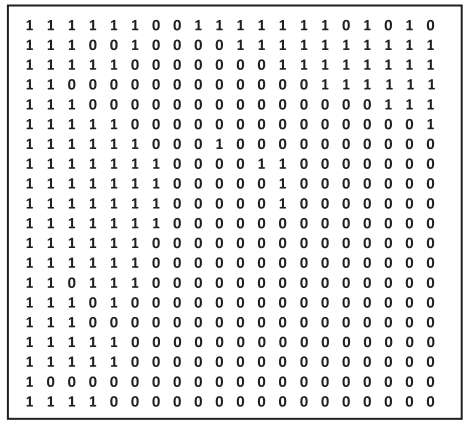
(2) Definitions of the grayscale and binary images (Coq generated)

(3) Proof of equivalence of the two versions of the grayscale and binary images

(4) Other operations over the image

When the RGB image is read, it is first translated to an RGB matrix where every group of three consecutive gray values corresponds to the three colors red, green, and blue of the pixel. The gray value of a pixel in the grayscale matrix is the average of the red, green, and blue colors' values of the corresponding pixel in the RGB image. The grayscale image matrix is converted to a formal definition of the grayscale image, both by the tool and by the Coq

(a)



(b)

Grayscale image matrix:

[97;93;84; . . . 136;137;138]

Grayscale image:

[G{0,0,97};G{0,1,93};G{0,2,84} . . .
;G{19,16,136};G{19,17,137};G{19,18,138}]

Binary image (threshold 120):

B{0,0,1};B{0,1,1};B{0,2,1} . . .
B{19,16,0};B{19,17,0};B{19,18,0}]

(c)

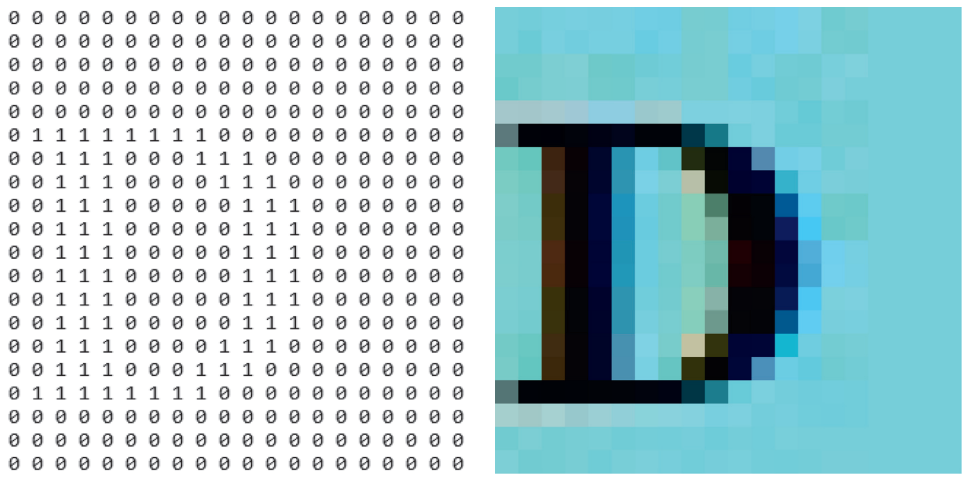
Figure 5: Example 2: translating the RGB image to matrix and Coq definition gsimage. (a) RGB image. (b) Binary image matrix. (c) Coq definitions of the image.

```
Fixpoint createimage (matrix: list nat) (row: nat) (col: nat) (maxcol: nat): gsimage \coloneq
match matrix with
  |nil ⇒ nil
  |cons gray m ⇒
    match row, col with
    |O, O ⇒ [Grow, col, gray]
    |S r, O ⇒ createimage m r maxcol maxcol) ++ [Grow, col, gray]
    |O, S c ⇒ (createimage m row c maxcol) ++ [Grow, col, gray]
    |S r, S c ⇒ (createimage m row c maxcol) ++ [Grow, col, gray]
    end
end.
```

Listing 29: Definition of function createimage.

```
Theorem gsimage_eq: coqgsimage = gsimagecoq.
Proof.
  unfold coqgsimage.
  unfold gsimagecoq.
  simpl.
  reflexivity.
Qed.
```

Listing 30: Proof of equivalence of grayscale images.



(a) (b)

Figure 6: Binary image matrix comparison with the corresponding RGB image (Example 1). (a) Binary image matrix. (b) RGB image.

(a) (b)

Figure 7: Binary image matrix comparison with the corresponding RGB image (Example 2). (a) Binary image matrix. (b) RGB image.

function createimage. In either case, the grayscale image conforms to the Coq formal definition gsimage (Listing 7). To verify that this translation is sound, we defined a function createimage (Listing 29) to generate a grayscale image from the matrix. The function createimage gets a matrix (list of gray values) and its dimensions (number of rows and columns) and generates a grayscale image of the form given in Figures 4 and 5. In the theorem gsimage_eq (Listing 30), we prove that the grayscale image coqgsimage generated by the tool is exactly the same as the image gsimagecoq generated by our formal function createimage.

The grayscale matrix is then converted to the formal definition of a binary image using a threshold value or a pair (range) of threshold values. Again, this translation is carried out by both the tool and the Coq function. The formal definition of the binary image (Listing 4) is a list of pixels where each pixel is a tuple of row, column, and binary value. The tuple, defined as a record type in Coq, has member rows, cols, and pixels as a list of nat. A recursive function imagetorealimage is defined to generate the tuple from the Coq binary image. To show that the translation to binary image is sound, a proof (Lemma binimage_eq) is automatically generated. This lemma states that the tool generated binary image coqbinaryimage and the binary image binimagecoq generated by the formal definition thresholding are the same.

In addition, the file proofofgsi.v has some fixed code for different operations over the image chosen for the analysis. These operations include thresholding (to generate binary image from the grayscale image), computing area size and run-length, and summing the run-lengths of the binary image. Currently, the images chosen are small (20×20 pixels). Larger images can also be translated, but the resulting binary images in the Coq would take longer (in order of minutes) to compile. The tool, though, can be easily

modified to split the image in slices and then perform the analysis for each slice separately.

We used our tool and converted multiple 20×20 RGB images to Coq script. Two 20×20 (400 pixels) translated RGB images with their Coq definitions of the matrix, grayscale, and binary forms are given in Figures 4 and 5 (the image matrix, Coq grayscale, and binary images are too long to be included here and are available at our repository). These grayscale images were created by the tool from their corresponding image matrices and then translated to Coq notations.

As mentioned earlier, the tool can generate binary image from the RGB image. The binary image matrix of the RGB image 4 is shown in Figure 6 (the RGB image from Figure 4 is resketched). Similarly, the binary image matrix of the RGB image 5 is shown in Figure 7. These binary images are retrieved using the threshold values 100 and 120, respectively. For each pixel of the binary image, the value, whether 0 or 1, is calculated by comparing the corresponding pixel's gray value with the threshold. The binary pixel is 0 if the pixel gray value is greater than the threshold; otherwise, it is 1. Note that the binary image matrix would change for different values of the threshold.

7. Conclusion

Binary images include pixels of just two gray values, black and white, and require only one bit per pixel storage. Binary image processing involves different transformations and encoding in a wide range of applications, including digital forensics and image integrity. To investigate binary image processing using formal techniques, a lightweight formal model of the binary images and multiple image transformations were defined in the Coq proof assistant. Using the interactive proof facility of Coq, several properties of the transformations were stated and proved. The formal system was tested against real RGB images though a prototype translator. The binary image defined as Coq notations is converted back to binary matrix.

Currently, the formal development can be tested against light (20×20 pixels) RGB images; however, high-resolution images would take longer (in order of minutes) to check. In the future, this issue may be resolved by splitting the high-resolution RGB images into slices. Each slice of the RGB image may be treated as a lightweight RGB image and processed separately. Furthermore, we formally specified and verified a number of interesting properties of the binary images and image transformations, but many are still missing. For example, the current formal system could be extended in the future to include proof of the distance properties of Euclidean and Chessboard distance functions.

Abbreviations

RGB: Red, green, and blue
HOL: Higher-order logic
CNN: Convolutional neural network
HDL: Hardware description language.

Data Availability

This research paper includes Coq formal developments and C++ code. All the source codes are available online at our repository at https://github.com/wilstef/binaryimagescoq.

Conflicts of Interest

The authors declare that they have no conflicts of interest regarding the publication of this paper.

Authors' Contributions

W.K. and S.I. carried out the formal specification and verification in the Proof Assistant Coq and wrote the first draft of the paper. W.K., A.R.A., and A.Q. contributed to the design and development of the C++ code and carried out the experiments. A.A., A.H., and S.A. reviewed the paper, contributed to the formal proof checking, and arranged the funding for this research.

Acknowledgments

This research was funded by the Deanship of Scientific Research at Prince Sattam Bin Abdulaziz University, Al-Kharj, Saudi Arabia.

References

[1] R. Chauhan, P. Mishra, and R. C. Joshi, "An overview of digital image forensics: image morphing and forgery detection algorithms," in *Information Security and Optimization*, pp. 107–120, Chapman and Hall/CRC, London, UK, 2020.

[2] S. Lai and R. Böhme, "Countering counter-forensics: the case of JPEG compression," in *International Workshop on Information Hiding*, pp. 285–298, Springer, 2011.

[3] P. S. Othman, R. R. Ihsan, R. B. Marqas, and S. M. Almufti, "Image processing techniques for identifying impostor documents through digital forensic examination," *Image Process. Tech*, vol. 62, pp. 1781–1794, 2020.

[4] G. Horsman and J. R. Lyle, "Dataset construction challenges for digital forensics," *Forensic Science International: Digital Investigation*, vol. 38, Article ID 301264, 2021.

[5] A. Shaus, Y. Gerber, S. Faigenbaum-Golovin, B. Sober, E. Piasetzky, and I. Finkelstein, "Forensic document examination and algorithmic handwriting analysis of Judahite biblical period inscriptions reveal significant literacy level," *PLoS One*, vol. 15, no. 9, Article ID e0237962, 2020.

[6] S. Agarwal, D. Sharma, and K. H. Jung, "Forensic analysis of colorized grayscale images using local binary pattern," in *Proceedings of the 2020 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, pp. 507–510, Noida, India, 2020.

[7] J. A. Redi, W. Taktak, and J.-L. Dugelay, "Digital image forensics: a booklet for beginners," *Multimedia Tools and Applications*, vol. 51, no. 1, pp. 133–162, 2011.

[8] R. Böhme and M. Kirchner, "Counter-forensics: attacking image forensics," in *Digital Image Forensics*, pp. 327–366, Springer, Berlin, Germany, 2013.

[9] Q. Duan, T. Akram, P. Duan, and X. Wang, "Visual saliency detection using information contents weighting," *Optik*, vol. 127, no. 19, pp. 7418–7430, 2016.

[10] I. D. Hernández, J. V. Hernández-Fontes, M. A. Vitola, M. C. Silva, and P. T. Esperança, "Water elevation measurements using binary image analysis for 2D hydrodynamic experiments," *Ocean Engineering*, vol. 157, pp. 325–338, 2018.

[11] Z. Zhao, C. Yao, C. Tang, C. Li, F. Yan, and S. Islam, "Diagnosing transformer winding deformation faults based on the analysis of binary image obtained from FRA signature," *IEEE Access*, vol. 7, pp. 40463–40474, 2019.

[12] H. Kim, J. Lee, E. Ahn, S. Cho, M. Shin, and S.-H. Sim, "Concrete crack identification using a UAV incorporating hybrid image processing," *Sensors*, vol. 17, no. 9, Article ID 2052, 2017.

[13] P. Narendran and J. Stillman, "Formal verification of the Sobel image processing chip," in *Current Trends in Hardware Verification and Automated Theorem Proving*, pp. 92–127, Springer, Berlin, Germany, 1989.

[14] N. G. Bourbakis and C. Alexopoulos, "A fractal-based image processing language: formal modeling," *Pattern Recognition*, vol. 32, no. 2, pp. 317–338, 1999.

[15] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks, Automated Technology for Verification and Analysis," in *Proceedings of the International Symposium on Automated Technology for Verification and Analysis*, pp. 269–286, Pune, India, 2017.

[16] P. Kouvaros and A. Lomuscio, "Formal verification of CNN-based perception systems," 2018, https://arxiv.org/abs/1811.11373.

[17] X. Sun, H. Khedr, and Y. Shoukry, "Formal verification of neural network controlled autonomous systems," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pp. 147–156, New York, NY, USA, 2019.

[18] P. Meredith, M. Katelman, J. Meseguer, and G. Roşu, "A formal executable semantics of Verilog," in *Proceedings of the Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pp. 179–188, Grenoble, France, 2010.

[19] M. Clavel, F. Durán, J. Hendrix, S. Lucas, J. Meseguer, and P. Ölveczky, "The Maude formal tool environment," in *Proceedings of the International Conference on Algebra and Coalgebra in Computer Science*, pp. 173–178, Bergen, Norway, 2007.

[20] W. Khan, D. Sanan, Z. Hou, and L. Yang, "On embedding a hardware description language in Isabelle/HOL," *Design Automation for Embedded Systems*, vol. 23, no. 3-4, pp. 123–151, 2019.

[21] W. Khan, A. Tiu, and D. V.F. Sanán, "An executable formal model of a hardware description language," in *Proceedings of the 2nd Singapore Cyber-Security R&D Conference*, pp. 19–36, Singapore, February 2017.

[22] W. Khan, M. Kamran, A. Ahmad, F. A. Khan, and A. Derhab, "Formal analysis of language-based android security using theorem proving approach," *IEEE Access*, vol. 7, pp. 16550–16560, 2019.

[23] T. Braibant and A. Chlipala, "Formal verification of hardware synthesis," in *Proceedings of the International Conference on Computer Aided Verification*, pp. 213–228, Saint Petersburg, Russia, 2013.

[24] J. Choi, M. Vijayaraghavan, B. Sherman, and A. Chlipala, "Kami: a platform for high-level parametric hardware specification and its modular verification," in *Proceedings of the ACM on Programming Languages*, pp. 1–30, New York, NY, USA, 2017.

[25] M. Bugliesi, S. Calzavara, R. Focardi, W. Khan, and M. Tempesta, "Provably sound browser-based enforcement of web session integrity," in *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium*, pp. 366–380, Vienna, Austria, 2014.

[26] W. Khan, S. Calzavara, M. Bugliesi, W. De Groef, and F. Piessens, "Client side web session integrity as a non-interference property," in *Proceedings of the International Conference on Information Systems Security*, pp. 89–108, Hyderabad, India, 2014.

[27] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan, "Automatic and robust client-side protection for cookie-based sessions," in *Proceedings of the International Symposium on Engineering Secure Software and Systems*, pp. 161–178, Munich, Germany, 2014.

[28] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan, "CookiExt: patching the browser against session hijacking attacks," *Journal of Computer Security*, vol. 23, no. 4, pp. 509–537, 2015.

[29] M. A. Alturki, J. Chen, V. Luchangco et al., "Towards a verified model of the Algorand consensus protocol in Coq," 2019, https://arxiv.org/abs/1907.05523.

[30] A. Anand, A. Appel, G. Morrisett et al., "CertiCoq: a verified compiler for Coq," in *Proceedings of theThird International Workshop on Coq for Programming Languages (CoqPL)*, Paris, France, 2017.

[31] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand, "CompCert-a Formally Verified Optimizing Compiler," in *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, 2016.

[32] A. Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*, MIT Press, Cambridge, MA, USA, 2013.

[33] B. C. Pierce, C. Casinghino, M. Gaboardi et al., "Software foundations," 2010, https://www.cis.upenn.edu/bcpierce/sf/current/index.html.

[34] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Springer Science & Business Media, Berlin, Germany, 2002.

[35] Y. D. Mistry, "Textural and color descriptor fusion for efficient content-based image retrieval algorithm," *Iran Journal of Computer Science*, vol. 3, no. 3, pp. 169–183, 2020.

[36] J. Wang and Y. Tan, "Efficient Euclidean distance transform algorithm of binary images in arbitrary dimensions," *Pattern Recognition*, vol. 46, no. 1, pp. 230–242, 2013.