UCLouvain

École polytechnique de Louvain

3D Watermarking for Additive Manufacturing

Author : **Léa LEBRUN**
Supervisor : **Benoît MACQ**
Readers : **Jérôme LECLÈRE**, **Aude SIMAR**
Academic year 2025–2026
Master [120] in Electro-mechanical Engineering

Acknowledgments

Finishing this thesis would not have been possible without the priceless help of these people.

I would first like to thank my supervisor, Prof. Benoît Macq, for his guidance throughout my thesis and for inviting me to take part in the Innovation classes for transition and sustainable development.

My sincere thanks go to Jérôme Leclerc for his precious advice, his help with my struggles, and his insights, which proved more than useful for my thesis. I am also grateful to Arthur Pisvin for his advice.

I am grateful to Prof. Aude Simar for being my reader and for her insightful mechanical lectures throughout my studies. My thanks also go to her and her team, especially Camille Van Der Rest, for their feedback and guidance.

I extend my gratitude to Régis Lomba for granting me access to the 3D printers at the Makilab and for sharing his expertise on them.

Special thanks to my boyfriend, Issambre, for his unconditional support and his help reviewing and improving my thesis.

Finally, I would like to warmly thank my family, especially my parents, as well as my friends and my two cats, for their support throughout this thesis and my entire academic journey.

Abstract

Additive manufacturing, and Fused Deposition Modeling in particular, have become widely accessible, raising new concerns regarding intellectual property rights, authentication and traceability (particularly when a recovered printed object is part of a firearm that contains no information indicating who printed it). In this context, 3D watermarking allows traceability information to be encoded directly into the part at the time of manufacture.

This Master's thesis proposes an approach to watermarking in which a locally generated opaque identifier is embedded in the infill during slicing, within PrusaSlicer, and retrieved from a reconstructed density volume without access to the original model. Two encoding methods have been implemented: the first involves tilting the orientation of the infill layers, and the second involves tilting the lines of the infill grid. A third method, which uses a key-dependent hash to determine the encoded bit on the line, is explored but not implemented. Decoding was validated on cubes printed in PLA on a Prusa MK4S using a CT scanner simulation tool (TomoPy) in place of a physical scanner. These tests enabled the encoding and decoding methods to be optimised, and their capacity, robustness, fidelity and security to be assessed. The first method achieved a BER of 0 % and a checksum pass rate of 100 % across all tests conducted, but is limited in terms of capacity. Whilst the second method considerably increases this capacity, allowing for redundancy, it does so at the cost of a more fragile decoding chain. Ultimately, the results show that the payload can be encoded into the infill during slicing and decoded from a reconstructed volume without being detected by an average observer.

Notice on AI and inspirations

In the process of writing this thesis, I used AI tools, specifically Claude (Anthropic). This tool helped me reformulate and translate my ideas, originally written in French, into clear and readable text. Even though it was involved in the writing, all ideas and text were first drafted by me, then reviewed to ensure they reflected my own thinking. All ideas regarding the watermarking (encoding and decoding) are solely the result of my own work. I also used Claude to speed up my understanding of the slicing software PrusaSlicer and to debug my code. Finally, I used it to understand how G-code works, in order to write the function `gcode_to_voxels.py` (Listing A.1). The structure of this thesis was inspired by [1].

Contents

Abbreviations

1D	1-Dimensional
3D	3-Dimensional
ABS	Acrylonitrile Butadiene Styrene
AM	Additive Manufacturing
BER	Bit Error Rate
CAD	Computer Aided Design
CRC	Cyclic Redundancy Check
CT	Computed Tomography
DMD	Direct Metal Deposition
EBM	Electron Beam Melting
ECC	Error-Correcting Code
FDM	Fused Deposition Modeling
GUI	Graphical User Interface
HD	Hamming Distance
IP	Internet Protocol
IPR	Intellectual Property Rights
MAC	Media Access Control
NAT	Network Address Translation
PETG	Polyethylene Terephthalate Glycol
PLA	Polylactic Acid
SLA	Stereolithography
SLM	Selective Laser Melting
STL	Standard Triangle Language
UCLouvain	Université Catholique de Louvain
UUID	Universally Unique IDentifier
XOR	Exclusive OR

Chapter 1

Introduction

Since the appearance of 3-Dimensional (3D) printing, the manufacturing of objects has been drastically improved. 3D printing has revolutionised industrial manufacturing, especially because it enables mass customisation and fast prototyping, and because it makes it possible to manufacture geometries that could not be produced with subtractive manufacturing. It also reduces production costs, manufacturing time and the storage space required.

But this rise of 3D printing comes with a problem: the technology has become widely accessible and easy to use for anyone [2, 3]. This accessibility raises concerns regarding Intellectual Property Rights (IPR), traceability and the authentication of manufactured parts. IPR are a concern because many models are protected, and 3D printing combined with 3D scanning now makes it possible to reverse-engineer printed models in order to replicate and share them [4]. Traceability is a concern because 3D printing can now be used to manufacture firearm components, so it becomes important to be able to determine who printed a given part [5, 6]. Finally, authentication is a concern because 3D printing makes it easy for anyone to modify a 3D model, even though some manufactured parts must comply with specific norms; a watermark could then be used to verify whether a part still complies with those norms.

A single type of solution can address this wide range of problems in different ways: 3D watermarking. It consists in embedding into the printed object a mark that carries some information: for IPR, the watermark acts like a copyright symbol; for traceability, it can contain information about the person or the machine that printed the object; and for authentication, it can be used to detect whether the object has been modified, since such a modification would corrupt the watermark. This thesis pursues the traceability use case: it embeds, during the slicing stage and directly within the slicing software PrusaSlicer [7], a mark that can later be recovered from the printed object alone without access to the original 3D model.

To understand why the slicing stage was chosen, it is useful to recall the pipeline of a 3D print. A 3D model is first created, either designed with Computer Aided Design (CAD) software or downloaded, typically as an Standard Triangle Language (STL) file. This file is fed into a slicing software, which converts it into a set of machine instructions, the G-code. The G-code is then sent to the printer, which decodes each instruction and executes it, producing the physical object layer by layer. Once the object has been printed, it can, if needed, be scanned in an attempt to recover the watermark. This pipeline offers several possible insertion points for a watermark: the 3D model itself, the G-code, or the physical object during printing. Inserting it within the slicer means the mark is embedded automatically whenever an object is sliced, without requiring any additional step from the person printing it.

Two requirements guide this choice. First, the watermark must be imperceptible: the printed object must look and behave exactly like an unmarked one to an average observer. Second, decoding must be blind: the watermark must be recoverable directly from the printed object without access to the original 3D model or G-code file.

Chapter 2 reviews the existing literature on 3D watermarking and additive manufacturing, and progressively narrows it down to the choices retained for this thesis: blind detection, insertion within the slicer, Fused Deposition Modeling (FDM)/Polylactic Acid (PLA) printing, and an opaque identifier as payload. Chapter 3 presents the resulting encoding and decoding scheme, developed through three successive methods, along with the payload structure and the validation protocol used to test it. Chapter 4 describes how these choices were implemented, both inside PrusaSlicer and in the Python toolchain used to simulate Computed Tomography (CT) acquisition and decoding. Chapter 5 reports and discusses the results obtained for the two implemented methods, evaluated against capacity, robustness, fidelity and security. Finally, Chapter 6 summarises the contributions of this thesis and outlines directions for future work.

Chapter 2

State of the Art

This chapter surveys the existing work to design a watermarking scheme for 3D-printed objects, and progressively narrows this landscape down to the concrete choices retained for this thesis.

Section 2.1 introduces Additive Manufacturing (AM) itself: its defining principle, the advantages that explain its growing accessibility, and the main processes found across the field, since the choice of process and material later constrains which watermarking techniques are even applicable.

Section 2.2 then turns to watermarking as such, distinguishing it from steganography, establishing the criteria against which any scheme is judged, and reviewing the existing literature on 3D watermarking, organised by the stage of the manufacturing pipeline at which the mark is inserted.

Building on this review, Section 2.3 commits to the insertion point, process, material, and payload retained for this thesis, and derives from them the constraints specific to the intended forensic use case: tracing a printed part back to the computer that sliced it, even from a damaged or partial fragment.

Section 2.4 addresses the symmetrical problem of decoding: which acquisition techniques can recover a printed object's internal structure, and which one is retained given the resources available for this work. Section 2.5 closes the chapter by summarising the decisions made along the way.

2.1 Additive Manufacturing

This first section explains what AM is, its benefits, and briefly describes some of its processes.

2.1.1 Definition of Additive Manufacturing

AM refers to all processes for manufacturing three-dimensional objects by successively adding material, layer by layer, based on a digital model – as opposed to subtractive manufacturing, which produces a part by removing material from a solid block (machining, milling, etc.) [8].

The process always begins with a digital model designed using CAD or obtained by other means (3D scanning, downloading). This model is then sliced into a series of thin layers, which the machine physically reconstructs by depositing, fusing or solidifying material (plastic, metal, resin, ceramic, etc.) according to the trajectory calculated for each layer.

Historically, AM was reserved for rapid prototyping in an industrial context. However, this family of processes has become increasingly widespread amongst the general public, driven by the emergence of low-cost printers. This development can be explained by a number of advantages that set AM apart from conventional processes.

Advantages of Additive Manufacturing

Here are the main advantages of AM:

- The first is **design freedom**: the absence of moulds or specialised tooling makes it possible to produce geometries that cannot be achieved through machining or casting [9].
- The second is **mass customisation**: each part can be individually adapted without significant reconfiguration costs, unlike conventional processes where customisation requires new tooling [10].
- The third is the **reduction in material waste**: material is only added where it is needed, unlike machining, which removes a significant portion of the starting block [11].
- Finally, AM enables **decentralised production**: an object can be manufactured locally as long as its digital file is available, without the need for a supply chain or dedicated industrial infrastructure.

It is precisely this last characteristic, that raises the central question of this dissertation: the very accessibility that makes 3D printing widely available for legitimate purposes also paves the way for uncontrolled uses, in particular the manufacture of weapons, as discussed in Chapter 1.

2.1.2 Additive Manufacturing Processes

In AM, a distinction is generally made between two main families of materials: polymers and metals. The energy required to transform raw material into a solid part varies depending on the process and the material: UV radiation or heat for polymers (photopolymerisation, thermal extrusion), laser beams or electron beams for metals. The main technologies for each of these families are described below [12].

Fused Deposition Modeling

FDM, illustrated in Figure 2.1, involves a thermoplastic filament (PLA, Acrylonitrile Butadiene Styrene (ABS), Polyethylene Terephthalate Glycol (PETG)...) being heated in a nozzle and deposited layer by layer along the path defined by the slicer. Each layer partially fuses with the previous one as it cools. This technique produces a modest resolution, but is accessible and inexpensive.

Stereolithography

Stereolithography (SLA), illustrated in Figure 2.2, is a process in which a tank of liquid photosensitive resin is polymerised point by point by a UV laser that scans the surface. The build platform moves down (or up, depending on the configuration) by one layer height with each pass. This technique offers very high precision and a good surface finish, but the resin is more fragile and more expensive than PLA.

Direct Metal Deposition

Direct Metal Deposition (DMD), illustrated in Figure 2.3, is a metal 3D printing process using concentrated energy deposition, which melts metal powder using a laser beam as it is ejected through a nozzle.

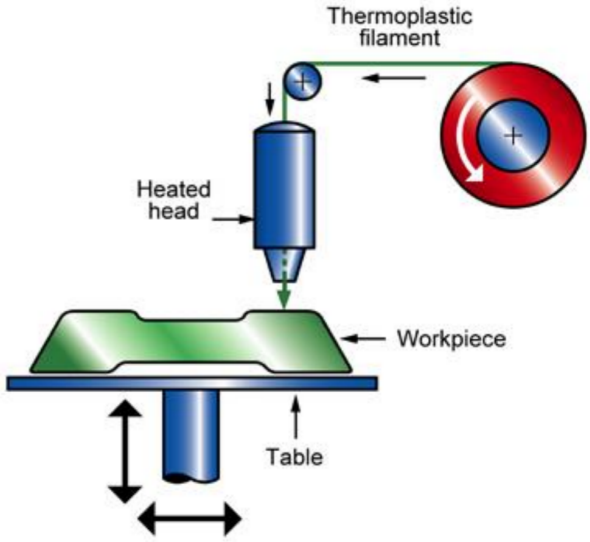


Figure 2.1: Principle of Fused Deposition Modeling (from [12])

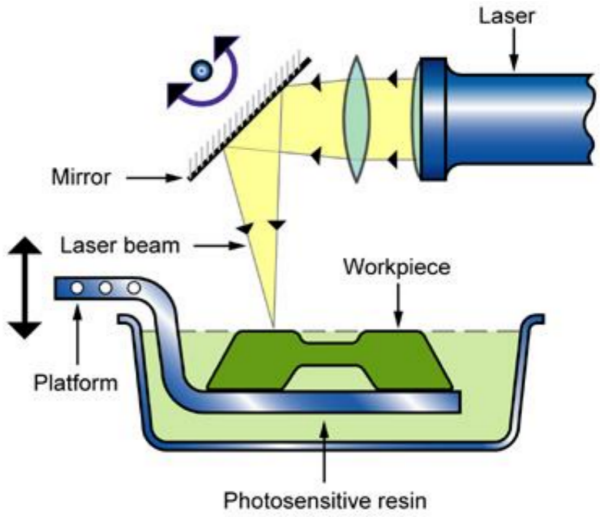


Figure 2.2: Principle of Stereolithography (from [12])

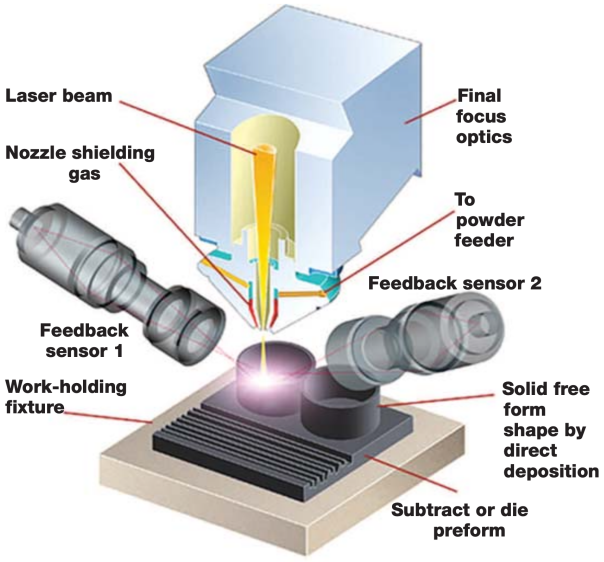


Figure 2.3: Principle of Direct Metal Deposition (from [12])

Selective Laser Melting

Selective Laser Melting (SLM), illustrated in Figure 2.4, is a process in which a high-power laser completely melts (rather than sintering) metal powder on a powder bed, layer by layer, in an inert atmosphere (argon/nitrogen) to prevent oxidation. This technique produces dense components with mechanical properties similar to those of solid metal.

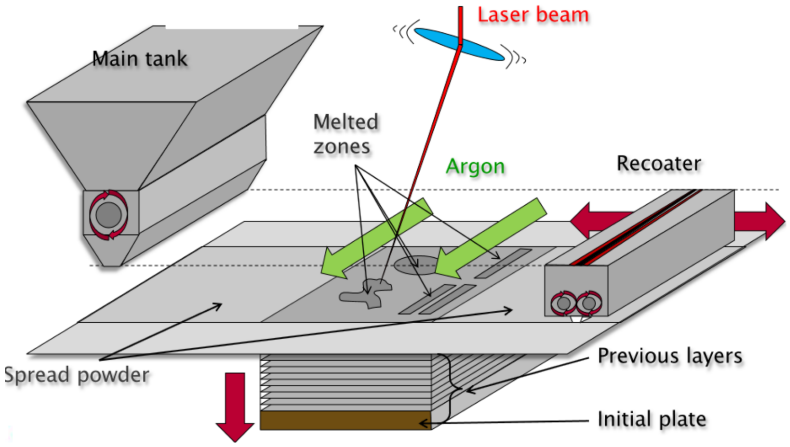


Figure 2.4: Principle of Selective Laser Melting (from [13])

Electron Beam Melting

In the case of Electron Beam Melting (EBM), illustrated in Figure 2.5, the same powder-bed logic applies, but fusion is achieved using an electron beam in a vacuum rather than a laser. The vacuum and the preheating of the powder bed reduce residual stresses. This technique is suitable for titanium alloys, for example.

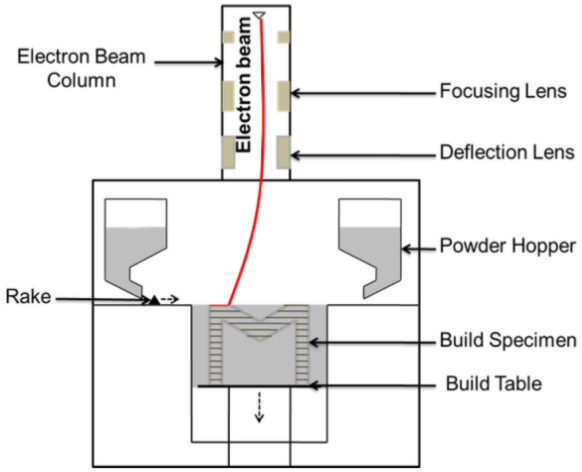


Figure 2.5: Principle of Electron Beam Melting (from [13])

The choice of process and material used for this master's thesis – FDM and PLA respectively – is explained in Section 2.3.2, once the point at which the watermark is inserted into the printing pipeline has been determined.

2.2 3D Watermarking

Having reviewed the additive manufacturing landscape, we now turn to the notion of a watermark itself: which criteria determine whether a watermarking scheme is effective, and how a watermark can be recovered once it has been embedded. We then examine the attacks (i.e., operations the watermarked object undergoes that may alter, degrade, or remove the watermark) a mark may be subjected to, and describe the complete workflow linking a digital model to a printed part. This sets up a review of the existing literature, organised according to the stage of the manufacturing pipeline at which the watermark is inserted.

2.2.1 Watermarking vs Steganography

Watermarking involves embedding information, the payload, into a piece of media (file, image, physical object, etc.) in such a way that it remains linked to the media even after transformations or attacks, whilst remaining imperceptible to an untrained user [14].

Watermarking differs from steganography in terms of its objective. Steganography aims primarily to conceal the communication channel itself: the mere fact of detecting that a hidden message exists already constitutes a failure of the system, regardless of whether it can be decoded. Watermarking, by contrast, is based on the principle that the existence of a mark may be known, or even public (as with banknotes), and focuses instead on its robustness: the mark must withstand attempts to remove or alter it whilst remaining linked to the original content [14].

This thesis focuses exclusively on watermarking: the aim is not to conceal the existence of a communication channel, but to embed a robust and verifiable mark within a printed object. In our case, we are interested in 3D prints; this is therefore referred to as 3D watermarking.

2.2.2 Watermark Evaluation Criteria

Here are the four criteria for assessing whether a watermarking method is effective [14]:

- **Capacity:** refers to the size of the payload (encoded information) that the system is capable of correctly holding. Generally, the payload is represented by a sequence of bits.
- **Robustness:** refers to the watermark's ability to withstand various attacks during the process. In our case, the main attacks will be printing, the use of the printed object, and scanning to detect the watermark (see Section 2.2.4).
- **Fidelity:** this involves embedding the watermark in a way that is imperceptible to the naked eye. A watermark with high fidelity makes it very difficult to distinguish between an unwatermarked image and a watermarked one.
- **Security:** a watermark is secure if it cannot be detected or decoded by an unauthorised person. Generally speaking, a malicious individual will attempt to remove the watermark.

It should be noted that a watermarking method cannot be both perfectly robust and faithful at the same time, there is always a trade-off between the two. Major changes to the geometry would make it more robust but much more visible and therefore less faithful. Conversely, changes that are too small would, admittedly, be imperceptible, but they would not withstand various attacks.

2.2.3 Detection Methods

As for how the watermark is decoded, there are three ways [14]:

- **Blind:** the watermark is decoded without the need for the original 3D model. Knowledge of the secret key and the watermarking algorithm is sufficient to detect or recover the watermark.
- **Non-blind:** in this case, the secret key and the watermarking algorithm alone are not sufficient to decode or detect the watermark, the original version of the media is required. This can also pose a security risk, as this original version must then itself be stored and protected.
- **Side-informed:** this is a middle ground between blind and non-blind scenarios, where the original version of the model is not required for detection, but only some information related to it.

In the context of this thesis, blind detection is the only viable option: a recovered artefact (for example, a fragment of a 3D-printed weapon) cannot be linked to any original digital model, as the latter was never saved or transferred outside the computer that carried out the slicing. The non-blind and side-informed scenarios are therefore, in fact, inapplicable to this use case, and this thesis focuses exclusively on a blind detection method.

2.2.4 Generic Attacks on 3D Models

The attacks most commonly analysed in the literature on 3D model watermarking are illustrated in Figure 2.6 using the Stanford Bunny model, a standard case study in the field [4]: rotation, additive noise, smoothing, coordinate quantisation, mesh simplification and subdivision, and cropping.

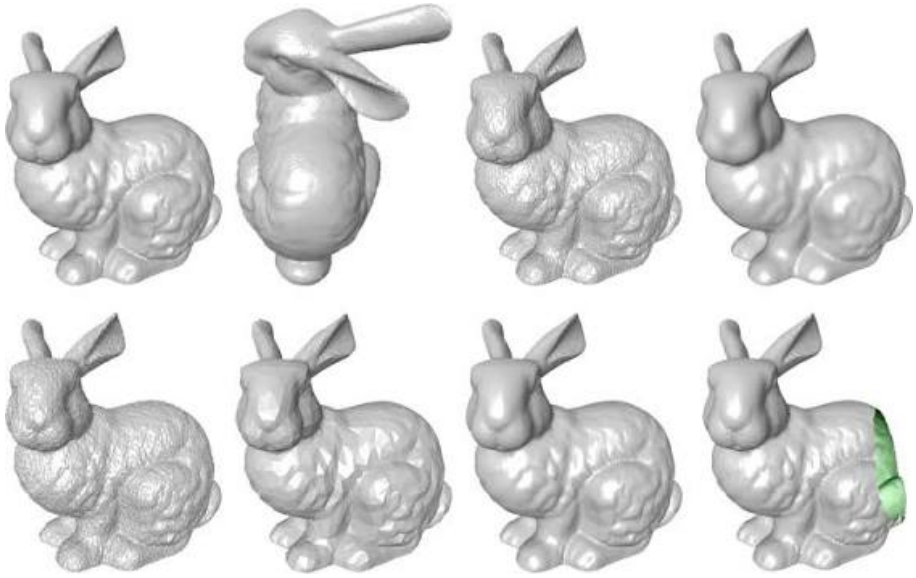


Figure 2.6: Attacks on the Stanford bunny model. Top left to top right: original model, rotation, additive noise, smoothing. Bottom left to bottom right: vertex coordinate quantisation, mesh simplification, mesh subdivision, cropping

Given this wide variety of attack types, robustness can be difficult to define. A watermark may be resistant to certain types of attacks and vulnerable to others, and there is no consensus in the literature on which attacks are the most critical to consider, the choice depends largely on the intended application context. Once we have defined our watermarking method, we will analyse the relevant attacks in Section 2.3.4.

2.2.5 Complete Workflow Description

The complete workflow leading from a digital model to a watermarked printed object, and then to its decoding, comprises the following steps (Figure 2.7):

1. **Digital model**: the object to be printed is either designed from scratch using CAD software, or obtained by downloading it from a 3D model-sharing platform. In both cases, the model is usually exported as an STL file, which describes the object's geometry as a mesh of triangles, without any information on how the object will actually be manufactured.
2. **Slicing**: this STL file is fed into slicing software (in this thesis, PrusaSlicer), whose role is to convert the geometric model into executable machine instructions. The slicer divides the model into a series of thin horizontal layers, determines the print orientation, generates support structures for overhangs where necessary, defines the outer walls (the perimeter) and fills the interior of the part according to a given pattern and density (the infill).
3. **G-code**: the result of slicing is a G-code file, a standardised format of machine instructions (movements, material extrusion, nozzle and bed temperature, speed, etc.) that the printer can execute line by line.
4. **Printing**: the printer's firmware interprets the G-code and controls the motors, the extruder and the heating elements to physically build up the object, layer by layer.
5. **Physical object**: the printed part can then be removed from its support, sanded down if necessary, assembled with other parts or used as it is.
6. **Recovery and scanning**: in the use case addressed by this thesis, the object is recovered retrospectively, without any information on the original digital model being available. The object is then scanned and reconstructed, either destructively or non-destructively (see Section 2.4).
7. **Decoding**: the decoding algorithm analyses this reconstructed structure to extract the payload encoded in the infill, without requiring the original model (blind detection; see Section 2.2.3).

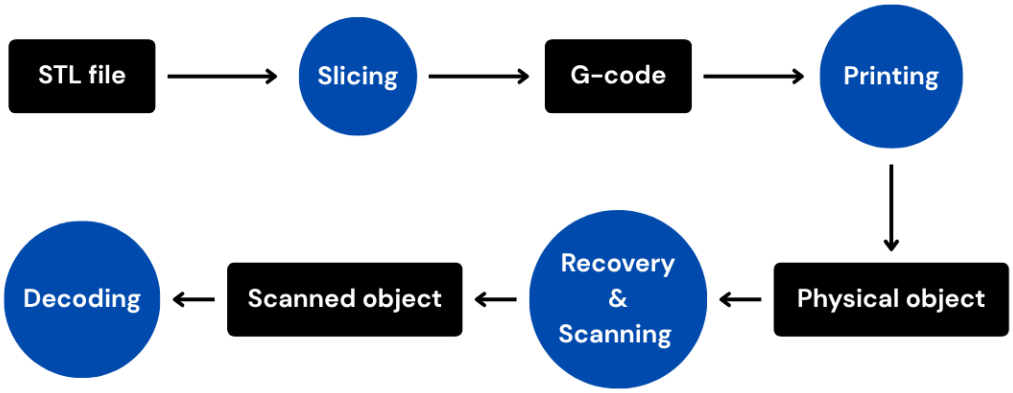


Figure 2.7: Complete workflow of the printing of a watermarked object

Watermarking can be applied during one of the first three stages of the pipeline, as we shall see in the next section.

2.2.6 3D Watermarking Schemes

The literature on 3D watermarking is organised here into two categories, based on the point at which the watermark is inserted into the manufacturing workflow (see Section 2.2.5).

3D Mesh Watermarking

One group of methods involves embedding the mark directly into the digital model itself – a triangular mesh or equivalent volumetric representation – before any conversion into machine instructions. These methods have the advantage of working independently of the slicer or printer used downstream, as the mark forms part of the input file.

This family has been the subject of a detailed survey [15], which reviews the signal processing techniques – geometric, spectral, multiresolution – underlying blind and robust mesh watermarking. [16] proposes a scheme of this type: each triangle in the mesh can be set to one of two possible geometric states (see Figure 2.10), with the choice of state encoding a bit of information; the method is robust to translations, rotations and scale changes, but has only been evaluated using simulated numerical attacks on the mesh, without any physical printing. [4] addresses the common weakness of this type of scheme when faced with cropping: their method identifies prominent characteristic points on the mesh, known as *prongs*, and repeats the watermark around each of them. This has also been tested solely in numerical simulation.

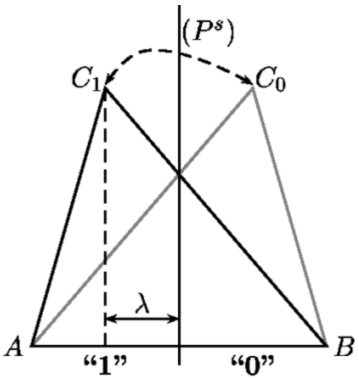


Figure 2.8: The two states of a triangle that embeds a bit (from [16])

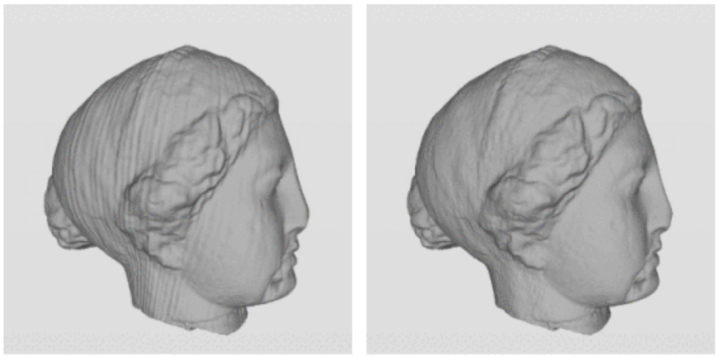


Figure 2.9: Sample of the watermarked models without visual masking (left) and with visual masking (right) (from [17])

Figure 2.10: Images to illustrate the 3D watermarking schemes

Other works in this category have been explicitly validated on actual printed objects. [17] inserts a conventional watermark into the mesh, but innovates in terms of decoding. The *layering artefact* (i.e., the staircase effect produced by layer-by-layer printing) serves as a reference point for estimating the printing orientation in a blind manner from an optical scan of the part, enabling the mark to be resynchronised and read without the original model (see Figure 2.10). [18], on the other hand, modifies the distribution of the mesh's surface normals – the distance between the surface and the model's centre of gravity – a mark designed to work with any slicer and any printer, decoded by a 3D scan of the printed part without requiring the original model.

The common feature of this family is the location of the insertion: prior to slicing, on the digital model. A mark inserted at this stage must survive conversion to G-code in order to remain legible after printing.

Watermarking During Slicing or in the G-code

A second family of methods does not act on the digital model, but directly on the slicer's output – the G-code, or an equivalent control file – without requiring access to the printer's firmware or the underlying model. This is the approach adopted for this thesis.

[19] locally modifies the layer thickness directly within the G-code generated by the slicer, across surface patches distributed throughout the part; the method is tested on actual FDM

prints and decoded in a single pass using a simple flatbed scanner (see Figure 2.11). [20] involves the formation of fine internal cavities whose arrangement encodes a binary code; the watermarked model is converted to G-code by a specialised slicer, and the cavities are read non-destructively using near-infrared transmission imaging on PLA parts. [21] adopts this same embedding principle and improves the reading of the cavities using video thermography, by stacking several successive thermal images to amplify the watermark signal.

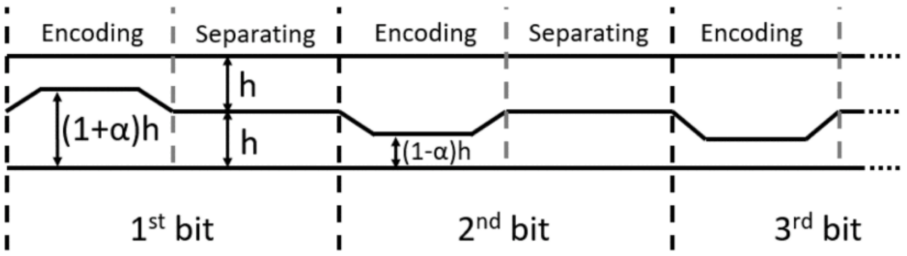


Figure 2.11: Encoding layer pattern

Fingerprinting

An alternative approach, distinct from active watermarking, involves identifying the printer that produced an object without deliberately embedding any information into it. [22] (PrinTracker) demonstrates that the hardware imperfections specific to each printer produce a signature distinctive enough to identify the source machine with 99.8% accuracy across a panel of fourteen printers. This approach differs fundamentally from watermarking: it exploits manufacturing noise that is already present, without active control over the encoded information or any guarantee of a structured payload.

Summary of the 3D Watermarking Schemes

Table 2.1 summarises the schemes reviewed above according to their insertion point, encoding mechanism, and validation status.

Ref.	Insertion	Mechanism	Detection	Acquisition	Validated on real print
[16]	Mesh	Binary geometric state per triangle	Not stated	—	No (numerical simulation)
[4]	Mesh	Repetition around characteristic points ("prongs")	Not stated	—	No (numerical simulation)
[17]	Mesh	Conventional mark + resynchronisation via the layering artefact	Blind	Optical scan	Yes
[18]	Mesh	Surface normal distribution	Blind	3D scan	Yes
[19]	G-code	Local layer-thickness modulation	Blind	Flatbed scanner	Yes
[20]	G-code	Internal cavities (binary pattern)	Blind (implied)	Near-infrared transmission imaging	Yes
[21]	G-code	Internal cavities (same principle)	Blind (implied)	Video thermography	Not stated

Table 2.1: Comparative summary of the 3D watermarking schemes reviewed

2.3 Selecting the Watermarking Scheme

This section commits to the concrete choices that follow from the review of existing methods. We first justify the insertion point retained for this thesis, and derive from it the software, hardware and material used throughout. We then examine which parameters exposed by the slicer could carry the watermark, and why the infill is retained over the alternatives considered. The next section narrows the generic attacks introduced in Section 2.2.4 down to those specific to the use case of this thesis. Finally, we define what the watermark actually encodes.

2.3.1 Watermark Insertion

Section 2.2.6 has organised the literature into two families according to where the watermark is inserted along the manufacturing pipeline described in Section 2.2.5: on the digital mesh, or on the slicer's output (during or after the slicing). This thesis adopts the second option: the watermark is embedded inside the slicer itself, as it generates the infill toolpath, rather than on the mesh beforehand or on the G-code afterwards.

Inserting the watermark on the digital mesh would require it to survive the slicing step untouched. Since blind detection is the only viable option for this thesis (Section 2.2.3), the watermark must in any case be recovered from the printed object alone; embedding it on a representation that may not survive the very process meant to produce that object is a risk.

Beyond the slicer, a watermark could in principle be embedded even later in the pipeline, at the level of the printer firmware, which ultimately interprets the G-code and drives the physical extrusion. This option is not feasible in the context of this thesis, as we do not have access to a printer's firmware. This thesis therefore directly modifies a slicing software, rather than the digital model beforehand or the printer firmware afterwards.

2.3.2 Process and Material Choices

Embedding the watermark at the slicing stage requires direct control over the software performing this step. This constraint has shaped several of the technical choices made in this thesis.

First, this level of control requires an open-source slicing software, whose source code can be modified to integrate the encoding algorithm. PrusaSlicer, developed by Prusa Research [23], meets this requirement [24].

Second, the choice of PrusaSlicer determines the printing hardware available for the experimental validation of this thesis: fortunately, the Makilab (the Université Catholique de Louvain (UCLouvain) fablab [25]) provides Prusa MK4S printers [26] (see Figure 2.12), which use FDM technology. Table 2.2 takes some of its features.

This choice nonetheless remains relevant with respect to the use case targeted by this thesis. Several recent studies document that illegally 3D-printed firearms are overwhelmingly manufactured using FDM on consumer-grade printers – the most widespread, least expensive and most accessible technology [5]. A population study conducted among 3D printing practitioners further confirms that PLA is among the most commonly used polymers [6]. FDM/PLA is therefore, in addition to being the combination made accessible by the Makilab's equipment, a choice representative of the population actually targeted by this thesis.

2.3.3 Usable Slicing Parameters

With the insertion point fixed at the slicing stage and the hardware fixed at FDM/PLA, the remaining question is which of the parameters controlled by the slicer could be locally modulated to encode a binary payload. Several candidates were considered, exploiting either the geometric or the thermal degrees of freedom available during printing.

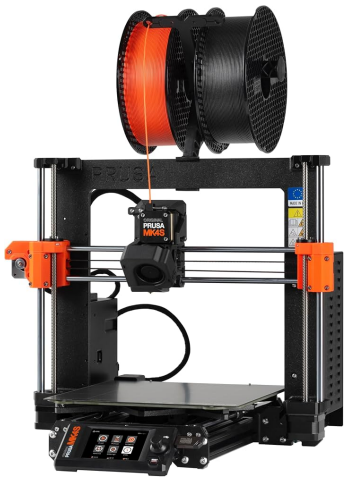


Figure 2.12: Prusa MK4S (from [26])

Features	
Build Volume	$250 \times 210 \times 220$ [mm]
Filament Diameter	1.75 [mm]
Layer Height	0.05–0.30 [mm]
Supported Materials	PLA, PETG, Flex, PVA, PC, PP, CPE, PVB

Table 2.2: Prusa MK4S features (from [27])

On the geometric side:

- **Layer thickness:** locally varying the height of individual layers, as already demonstrated in the literature [19].
- **Perimeter path deviation:** introducing small deviations in the trajectory followed by the outer wall could encode information geometrically. This acts directly on the outer, visible surface of the part, and any deviation large enough to be reliably read back could also be large enough to be seen.
- **Local over- or under-extrusion:** deliberately extruding slightly more or less material at chosen points along the perimeter would create small surface irregularities encoding information. This shares the same weakness as the previous option and additionally risks locally weakening the part or degrading its surface finish.

On the thermal side:

- **Nozzle temperature modulation:** varying the nozzle temperature to influence local extrusion density was also considered. This option was set aside as impractical: hot-ends have thermal inertia, so heating or cooling a nozzle to a new target temperature takes longer than printing a single small encoding region.

There exists another option that does not act on the outer geometry or on the thermal profile, but on the infill: the internal lattice structure a slicer generates to partially fill an otherwise hollow part. This could be done in several ways, for instance by displacing a point of the infill pattern, by reorienting a single element of it, or by reorienting the pattern as a whole. The exact mechanism retained for this thesis is developed in the next chapter (see Chapter 3).

Unlike the geometric and thermal options above, a mark embedded in the infill is entirely enclosed by the perimeters and top/bottom solid layers once printing is complete, and is therefore invisible on the finished object – the strongest possible fidelity, in the sense of Section 2.2.2. This advantage comes with a caution specific to FDM: the infill of the layer currently being printed is visible for as long as it is not yet covered by the next layer or by a solid top surface. Anyone directly observing the print in progress could, in principle, notice an unusual pattern; the encoding therefore still needs to remain visually discreet during printing, even though it is undetectable once the object is finished and handed over.

2.3.4 Constraints Specific to this Thesis

Since watermarking takes place during slicing, this thesis focuses on the attacks that occur during printing, during the use of the part, and during its acquisition through scanning for decoding purposes.

During printing, the FDM process itself introduces a degree of geometric imprecision inherent to layer-by-layer material deposition such as local extrusion variations, nozzle dimensional tolerance, imperfect inter-layer adhesion. These imprecision are the ones that the encoding method must tolerate without compromising the readability of the watermark.

During the use of the part, several post-processing operations may be applied: sanding, painting, gluing or assembling several separately printed parts, or simply mechanical wear from use.

The most decisive constraint for this thesis, however, concerns fragmentation. The watermark cannot rely on the availability of the entire part: it must be redundantly repeated across multiple regions of the infill, so that a sufficiently large fragment, even a partial one, still contains at least one complete and decodable copy of the payload.

Finally, during the scanning of a recovered part, the acquisition process reconstructs an approximation of the part's actual internal structure: reconstruction noise, limited resolution, and artefacts specific to the imaging method used must also be accounted for as sources of attack on the watermark.

2.3.5 Payload Content

The payload embedded by the watermark must serve a double purpose: it must allow a printed part to be traced back to the computer that sliced it, and it must remain decodable. Two natural candidates for its content were considered first, and both were ultimately discarded.

Embedding the Internet Protocol (IP) address of the slicing computer was an early candidate but it does not reliably identify a single machine: most internet service providers now rely on carrier-grade Network Address Translation (NAT) to share a single public IP address among many subscribers simultaneously, precisely because of IPv4 address scarcity. Recovering the specific subscriber behind a shared address requires the port number and a precise timestamp in addition to the address itself; even then, only the service provider holds the internal mapping needed to resolve this information to a specific subscriber, typically released only in response to a legal request [28].

The Media Access Control (MAC) address of the slicing computer's network interface was considered for the same reason, but suffers from an equivalent instability. Modern operating systems now widely implement MAC address randomisation to protect user privacy on Wi-Fi networks, so the address visible externally changes over time and no longer reliably identifies the same physical machine [29].

This thesis instead retains a locally generated, opaque identifier: a fixed-length random value drawn once, the first time the modified slicer runs on a given machine, and reused for every subsequent print. No central registry is needed – the same principle already used to generate a Universally Unique IDentifier (UUID) without any coordinating authority [30]: uniqueness is not absolute but probabilistic, and becomes overwhelmingly likely as long as the identifier is long enough relative to the number of machines that will ever generate one. The exact length retained for this thesis, together with the rest of the payload's bit budget, is derived in Section 3.2.

Because the identifier is opaque, attributing it to a person is kept separate from the watermark's technical design, and depends on how the software is used:

- If the slicer is linked to a Prusa Connect account [31], the identifier can be associated with that account server-side, at the time it is generated.
- If the slicer is used without an account, no such association exists remotely: attribution then becomes possible only by later examining the originating machine itself, comparing the identifier it holds locally against the one decoded from a recovered part.

2.4 Decoding Techniques

This section addresses the last remaining step of that workflow: acquiring enough information about its internal structure to make the payload embedded in the infill available to a decoding algorithm. Since detection must remain blind (Section 2.2.3), this reconstruction has to be obtained from the object itself. We review the acquisition techniques available for this purpose, before justifying the one retained for this thesis.

2.4.1 Overview of Acquisition Techniques

Surface optical scanning techniques, such as structured-light scanning [32] or thermography [33], offer a fast, inexpensive and reasonably accurate reconstruction of an object's external geometry. These techniques remain, however, limited to the surface or near-surface of the part: they provide no access to an internal structure such as the infill, which makes them unsuitable for reading the watermark developed in this thesis.

X-ray CT, by contrast, reconstructs a complete density volume of the part, and is therefore the only non-destructive modality considered capable of accessing the internal structure of the infill. Its main drawback is the cost and limited accessibility of the required equipment.

A destructive alternative consists of cutting the part along a single plane, for example, bisecting it to expose a cut face that crosses a large number of infill layers at once. This face can then be imaged in several ways: by photography, or with a 3D scan of the exposed face. A single cut remains simple and inexpensive, at the cost of destroying the part and maybe losing a part of the watermark.

Finally, a simulation on a known density matrix, using the TomoPy toolkit [34], allows the decoding algorithm to be validated independently of physical acquisition itself. This approach is not a field-deployable acquisition technique, but a validation tool, whose use in this thesis is detailed in Section 3.3.2.

2.4.2 Selected Decoding Scheme

Real CT scanning is the acquisition modality best suited to reading a watermark encoded in the infill, since it is non-destructive and gives access to the complete volume of the part. However, access to a real CT scanner could not be obtained within the time available for this work.

This thesis therefore adopts a simulation-based validation approach: the TomoPy toolkit generates projections from a known 3D density matrix, representing the watermarked part as produced by slicing, and then reconstructs this matrix from the simulated projections, in a manner analogous to a real CT scanner. This approach validates the logic of the decoding algorithm, i.e., it verifies that the payload can indeed be recovered from a reconstructed volume.

This approach carries a limitation that should be explicitly acknowledged: the simulation does not reproduce the full range of noise and distortion sources found in a real CT acquisition (detector noise, reconstruction artefacts, limits of effective resolution, geometric imprecisions of the printed part itself). Validating the decoding algorithm on a real CT scan of an actually printed part is therefore left as a direction for future work.

2.5 Chapter 2 Summary

This chapter has led to the following decisions:

- **Detection model:** blind detection is the only viable option (Section 2.2.3).
- **Insertion point:** the watermark is embedded inside the slicer itself (Section 2.3.1).
- **Process and material:** FDM and PLA, using PrusaSlicer and the Prusa MK4S printers available at the Makilab (Section 2.3.2).
- **Encoding channel:** the infill will be used to carry the payload (Section 2.3.3).
- **Payload:** a locally generated, opaque identifier, rather than an IP or MAC address (Section 2.3.5).
- **Validation strategy:** in the absence of access to a real CT scanner, decoding is validated through TomoPy simulations on a known density matrix (Section 2.4.2).

Chapter 3

Methodology

Chapter 2 narrowed the design space down to a small set of committed choices: blind detection, insertion inside the slicer itself, FDM/PLA on a Prusa MK4S, the infill as the encoding channel, and a locally generated opaque identifier as payload. What remains open is how exactly the infill's geometry is modulated to carry that payload, and how the payload is recovered from a reconstructed density volume without access to the original model.

Section 3.1 presents the encoding and decoding scheme itself, developed through three successive iterations: a first method that modulates the orientation of entire infill layers (Section 3.1.1), a second method that modulates individual grid lines to increase capacity (Section 3.1.2), and a third, still-partial method that replaces positional bit indexing with a keyed hash to add security and shape independence (Section 3.1.3). Section 3.1.4 compares the three along the criteria introduced in Section 2.2.2.

Section 3.2 defines the concrete bit layout of the payload and how decoding errors are detected and corrected. Finally, Section 3.3 describes the simulation pipeline used to validate decoding without a physical CT scanner, maps it onto the four evaluation criteria, and states the limitations carried forward from this chapter.

3.1 Design of the Encoding and Decoding Scheme

Section 2.3.3 retained the infill as the channel carrying the watermark. What that section left open is how the infill should be modulated to encode a bit. Two working methods and a third in progress are presented below through their encoding rule, their decoding rule, and a discussion against the four evaluation criteria of Section 2.2.2.

A cube with a side length of c is used for the test geometry throughout this chapter: its simple, constant cross-section keeps the encoding and decoding logic easy to develop and debug before tackling more complex shapes.

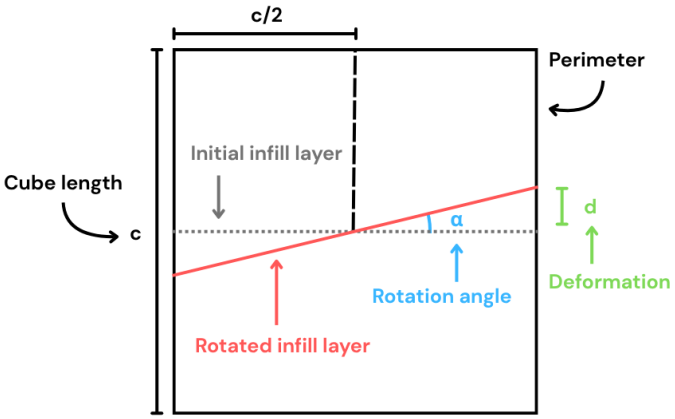
3.1.1 Method 1: Whole-Layer Orientation Modulation

Encoding

Method 1 encodes one bit per infill layer: to encode a '0', the layer's infill angle is left unchanged. To encode a '1', the whole layer is rotated by a fixed angle α about the centre of the infill region. The first infill layer always encodes a '0' and serves as a decoding reference (see Section 3.1.1). The watermark is repeated as long as layers are available.

Because the rotation pivots on the infill's centre, the point of maximum displacement between two differently oriented layers is at the perimeter (see Figure 3.1), where the infill meets the

printed wall. The angle of rotation α is therefore sized so that this maximum displacement equals a target value d:



$$\alpha = \arctan\left(\frac{2d}{c}\right) \tag{3.1}$$

Figure 3.1: Drawing of the rotation of the infill

As mentioned in Section 2.3.2, this thesis uses a Prusa MK4S printer from the Makilab. To characterise its positioning precision, calibration cubes were printed and measured; Table 3.1 reports the results for 2 cm-side cubes.

X-axis results [mm]	Y-axis results [mm]
20.08	20.00
20.04	20.00
20.10	20.00
20.02	20.00

Table 3.1: Results of the calibration test of a Prusa MK4S from Makilab

The largest error measured on these calibration cubes is 100 μm, on the X-axis (mean error: 60 μm); the Y-axis showed no measurable error over the same cubes. This asymmetry aside, a value for the target deformation d needs to satisfy two opposing constraints.

On one hand, d should not be much smaller than the printer's own positioning error: a deviation well below the printer's inherent imprecision would be difficult to distinguish reliably from ordinary printing noise once decoding is attempted. On the other hand, d should stay close to the limit of naked-eye perceptibility, so that the mark remains inconspicuous: from [35], standard (20/20) visual acuity resolves features of about 70–90 μm at a reading distance of 25 cm. The worst-case printer error of 100 μm sits right at this boundary – large enough to remain decodable given the printer's own noise floor, and close enough to the eye's resolution limit to stay effectively imperceptible, which is why it was retained as the target value for d throughout this chapter. For $c = 20$ mm and $d = 100\,\mu$m, Equation (3.1) gives $\alpha \approx 0.573°$.

Decoding

For decoding, the difference between each pair of consecutive infill layers $(i-1, i)$ is computed, and its mean value is taken:

$$\overline{\Delta}_i = \text{mean}(\text{layer}_{i-1} - \text{layer}_i) \tag{3.2}$$

A small $\overline{\Delta}_i$ indicates that layers $i-1$ and i share the same orientation (same bit); a large $\overline{\Delta}_i$ indicates a change of orientation, i.e. a bit different from the previous layer. Since the first infill

layer is always set to '0' to serve as a reference, the full bit sequence is recovered by propagating this relative decision layer by layer.

Discussion

With one bit encoded per infill layer, capacity is bounded by the number of infill layers: 104 bits for a 2 cm cube at a 0.2 mm layer height. Since the payload is 80 bits long (see Section 3.2.1), at least 241 infill layers ($3 \times 80 + 1$ for the reference layer) would be needed to have redundancy and perform a majority vote. Because the rotation pivots on the infill's centre, the perimeter is where two differently-oriented layers differ the most. The bit-to-layer mapping is fixed and public, offering no security at this stage. Combined with the capacity limit above, this motivated Method 2.

3.1.2 Method 2: Per-Line Grid Modulation

Encoding

Method 2 encodes one bit per infill line rather than per layer, which requires generating the infill geometry directly rather than perturbing the slicer's default pattern. A dedicated grid infill was implemented: two families of straight, perpendicular lines (vertical lines at constant x ("X lines") and horizontal lines at constant y ("Y lines")). A grid was retained as the underlying infill pattern because straight lines are simpler to generate and to modulate individually than the elements of a more complex pattern, and because decoding a straight-line grid is correspondingly simpler than decoding a triangle mesh for example.

Within a layer, encoding proceeds X lines first (left to right), then Y lines (bottom to top). A line encodes a '1' by being tilted by a small angle around its own centre, following the same sizing principle as Equation (3.1), with the line's own length used in place of c so that the maximum displacement stays bounded by the same 100 μm target regardless of line length; it encodes a '0' by being left at its nominal orientation (90° for X lines, 0° for Y lines). The first and second layers of the infill are used as a reference for the decoding. The watermark is repeated as long as lines are available. Figure 3.2 illustrates this method's encoding principle for the following payload: [1,1,0,1,1,0,1,0].

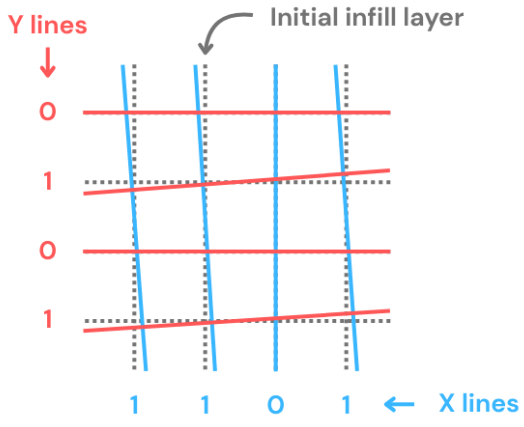


Figure 3.2: Illustration of Method 2 encoding principle

The encoding scheme also has to remain compatible with slicers that generate infill for different layers concurrently rather than strictly sequentially. Bit positions are therefore computed as a pure function of a line's layer and index.

Decoding

Decoding compares each layer's profile against a reference built from two dedicated calibration layers, the first infill layer forced to bit '0' on every line and the second forced to bit '1'. Summing, column-wise and row-wise, the difference between these two calibration layers yields two 1-Dimensional (1D) reference profiles with a peak at every line position: since one calibration layer is fully untilted and the other fully tilted, every line contributes a peak, while everything else cancels out. For every subsequent layer, the same column-wise and row-wise summation is applied to the difference between that layer and the untilted calibration layer, producing a candidate profile of the same shape; within each reference peak's index range, a bit is read as '1' if the candidate profile's mean value over that range is close to the reference profile's (the line differs from the untilted calibration as much as a fully tilted line would), and as '0' if it collapses toward zero (the line stayed close to the untilted calibration).

Discussion

Capacity is $(N_x + N_y)$ bits per layer, where N_x and N_y are the number of X and Y lines set by the infill area and density, multiplied by the number of layers: 816 bits for a 2 cm cube at a 0.2 mm layer height (104 layers × 8 lines/layer - 16 lines used for the reference layers) – well above Method 1's capacity for the same geometry, and large enough to repeat the payload for redundancy (Section 3.2). The bit-to-line mapping remains fixed and public, so security is still absent, and decoding still assumes a constant cross-section along the object's height; Section 3.1.3 addresses both points together.

3.1.3 Method 3: Hash-Based Line Indexing

This method could not be implemented within the scope of this thesis but is presented as a proposal for future work.

Encoding

Instead of assigning bit positions by counting layers or lines in a fixed order – which requires a constant cross-section and offers no security – Method 3 computes, for each candidate line at position (x, y, z), an index idx and a flip bit $flip$ from a keyed hash of that position and a secret key K:

$$(idx, flip) = h(x, y, z, K) \tag{3.3}$$

The line is tilted (encodes a geometric '1', following the same angle sizing as Methods 1–2) if $payload[idx] \oplus flip = 1$, and left at its nominal orientation otherwise. Any number of lines can map to the same idx, which gives redundancy.

Decoding

After reconstruction, the decoder recomputes $(idx, flip)$ for every detected line from its position and the shared key K, undoes the flip, and accumulates a majority vote for $payload[idx]$ over every line that hashed to that index.

Example

In order to have a better understanding of this method, here is a concrete example. Consider a 4-bit payload $payload = [1, 0, 1, 1]$ (indices 0 to 3) and seven infill lines. For each line, the hash of its position and the secret key K yields the index and flip bit shown in Table 3.2. Encoding then follows state = $payload[idx] \oplus flip$ (tilted if 1, nominal if 0). Lines 1 and 5 both carry

Line	idx	flip	payload[idx]	Encoded state	Decoded vote
1	0	1	1	nominal	1
2	1	0	0	nominal	0
3	2	0	1	tilted (*misread as nominal*)	0
4	2	1	1	nominal	1
5	0	0	1	tilted	1
6	3	1	1	nominal	1
7	2	0	1	tilted	1

Table 3.2: Worked example of Method 3's encoding and decoding on a 4-bit payload and line 3 is misread during decoding.

payload bit 0 (index 0), yet print differently (one nominal, one tilted) since their flip bits differ, without knowing K nothing suggests these two lines are related.

At decoding, each line's $(idx, flip)$ is recomputed from its position and K, and the observed state is un-flipped to obtain a vote for $payload[idx]$ (*Decoded vote* column). Index 0 receives two votes (Lines 1 and 5, both 1); index 1 receives one vote (Line 2: 0); index 2 receives three votes (Lines 3, 4 and 7: 0, 1, 1); index 3 receives one vote (Line 6: 1). Taking the majority at each index recovers exactly the $payload = [1, 0, 1, 1]$, including at index 2, where the majority (two votes out of three) overrides the single vote corrupted by Line 3's misread.

Discussion

Because it reuses the same per-line geometric encoding as Method 2, it has the same capacity. Unlike Methods 1 and 2, the bit-to-line mapping now depends on a secret key K: without it, an attacker cannot predict which lines carry which bits, making Method 3 the first of the three to offer any security.

3.1.4 Comparison of the Three Methods

Table 3.3 summarises the three methods along the insertion mechanism, capacity, generality with respect to infill pattern and object shape, and security.

3.2 Payload Structure and Error Handling

3.2.1 Payload Sizing

Section 2.3.5 retained a locally generated, opaque identifier as payload, drawn once and reused for every print, rather than an IP or MAC address. Since this identifier is generated independently by every machine, with no central registry, its length must be chosen so that two different machines are very unlikely to ever draw the same value.

For k machines each drawing an n-bit identifier uniformly at random, the probability that at least two of them draw the same value is approximated using the birthday paradox formula [36]:

$$P(\text{collision}) \approx 1 - \exp\left(-\frac{k^2}{2 \cdot 2^n}\right) \tag{3.4}$$

Choosing n therefore requires an estimate of k, the number of machines that could ever run this watermarking scheme. Desktop FDM 3D printers reached approximately 1.5 million units

	Method 1	**Method 2**	**Method 3**
Insertion mechanism	Whole infill layer	Individual grid line	Individual grid line
Bit-to-position mapping	Fixed (layer order)	Fixed (line order)	Keyed hash
Capacity	≈Number of layers	≈ $(N_x+N_y)\times$ number of layers	≈ $(N_x+N_y)\times$ number of layers
Works with any infill pattern	Yes	No (custom grid only)	No (custom grid only)
Works with varying cross-section	No	No	Yes
Redundancy	Possible (at least 241 layers needed)	Possible	Possible
Security	None	None	Key-dependent

Table 3.3: Comparison of the three watermarking methods designed for this thesis

sold worldwide in 2023, a figure still growing year-on-year [2], with more than one million units sold in the first quarter of 2025 alone [3]. Table 3.4 reports the resulting collision probability, from Equation (3.4), for three deliberately conservative estimates of k against several candidate values of n.

Machines k	$n = 40$	$n = 48$	$n = 56$	$n = 64$
1.5 M (2023 global sales)	64.1%	0.399%	1.56×10^{-3} %	6.10×10^{-6} %
5 M (current annual rate)	>99%	4.34%	0.0173%	6.78×10^{-5} %
50 M (10-year cumulative estimate)	>99%	98.8%	1.72%	6.78×10^{-3} %

Table 3.4: Probability of at least one collision among k machines, for identifiers of length n, computed from Equation (3.4)

At $n = 40$ bits, a single year of global FDM printer sales is already enough to make a collision more likely than not. $n = 64$ bits, by contrast, keeps the collision probability around 10^{-3} under the 10-year estimate. This thesis therefore retains a 64-bit opaque identifier.

3.2.2 Payload Layout

Each copy of the payload is structured as:

```
[ opaque identifier (64 bits) ][ checksum (16 bits) ]
```

The checksum is a fixed-size value computed from the identifier and verified at decoding time, used to tell whether a given copy of the identifier was read correctly. A Cyclic Redundancy Check (CRC), the checksum used in this design, uses several check bits computed through a fixed binary division to catch a wide range of errors.

Concretely, an r-bit CRC is computed by treating the identifier and a fixed, known generator polynomial G of degree r as binary polynomials. The identifier is first padded with r zero bits, and this padded value is divided by G using modulo-2 (Exclusive OR (XOR)-based) polynomial division [37]. The r-bit remainder of that division is the checksum, which replaces the padding zeros. At decoding, the same division is applied to the received (identifier + checksum) value using the same G: a non-zero remainder flags the copy as corrupted, while a zero remainder is a probabilistic sign that the copy is intact.

Checksum Sizing

Choosing the checksum's length r trades off two opposing costs. On one hand, a longer checksum catches more corrupted copies. On the other hand, every one of those r bits is added to *every* copy of the payload embedded in the part, directly competing with the redundancy that Methods 2 and 3 rely on for robustness.

An r-bit checksum's detection strength is characterised by its *Hamming Distance* (Hamming Distance (HD)): the minimum number of simultaneously corrupted bits that can produce an undetected error, for a given polynomial and message length [38].

In Figure 3.3 [38], rows correspond to increasing Hamming Distance (HD) and columns to CRC sizes from 3 to 16 bits. Each cell gives, for that combination of size and target HD, the maximum message length still covered by that guarantee, together with the polynomial (in hexadecimal) that achieves it.

Max length at HD / *Polynomial*	CRC Size (bits)													
	3	4	5	6	7	8	9	10	11	12	13	14	15	16
HD=2	2048+ 0x5	2048+ 0x9	2048+ 0x12	2048+ 0x21	2048+ 0x48	2048+ 0xA6	2048+ 0x167	2048+ 0x327	2048+ 0x64D	–	–	–	–	–
HD=3		11 0x9	26 0x12	57 0x21	120 0x48	247 0xA6	502 0x167	1013 0x327	2036 0x64D	2048 0xB75	–	–	–	–
HD=4			10 0x15	25 0x2C	56 0x5B	119 0x97	246 0x14B	501 0x319	1012 0x583	2035 0xC07	2048 0x102A	2048 0x21E8	2048 0x4976	2048 0xBAAD
HD=5						9 0x9C	13 0x185	21 0x2B9	25 0x5D7	53 0x8F8	none	113 0x212D	136 0x6A8D	241 0xAC9A
HD=6							8 0x13C	12 0x28E	22 0x532	27 0xB41	52 0x1909	57 0x372B	114 0x573A	135 0xC86C
HD=7									12 0x571	none	12 0x12A5	13 0x28A9	16 0x5BD5	19 0x968B
HD=8										11 0xA4F	11 0x10B7	11 0x2371	12 0x630B	15 0x8FDB

Figure 3.3: Table 3 from [38]

At our identifier's length (64 bits), an 8-bit CRC can only sustain HD 4 (valid up to 119 bits, polynomial `0x97`), since HD 5 already drops to a 9-bit limit. A 16-bit CRC, by contrast, can still sustain HD 6 at this length (valid up to 135 bits, polynomial `0xC86C`) – guaranteeing detection of every error pattern affecting up to 5 bits simultaneously. On that basis, this thesis retains a 16-bit CRC (`0xC86C`). Each payload copy is therefore 80 bits long in total.

3.2.3 Error Correction Strategy

Rather than a formal Error-Correcting Code (ECC) [39], this thesis relies on redundancy through repetition and majority voting. Both Methods 2 and 3 provide this redundancy. The checksum defined in Section 3.2.2 plays two distinct roles in this scheme.

- For Method 2's literal repetition: since the payload copies occupy fixed consecutive positions in a bit stream read in a known order, a corrupted copy can be identified and discarded before it is allowed to influence the vote. Method 3 has no equivalent notion of a discrete copy to discard, since its redundancy is distributed bit by bit across independently hashed lines.
- Once a payload has been assembled from the surviving votes, its checksum offers a final consistency check on that result. A checksum that fails on the final assembled payload is a signal that the result should not be trusted.

3.3 Validation Protocol

Since no real CT scanner could be accessed within the time available for this thesis, decoding is instead validated by simulating what such a scan would produce. The pipeline has two stages: converting a watermarked part's G-code into a voxelised volume, and feeding that volume through a simulated CT scan (TomoPy).

3.3.1 From G-code to a Voxelised Volume

A voxel, the three-dimensional equivalent of a pixel, is a small cube of fixed size carrying a single value; here, that value indicates whether the voxel is empty (0) or filled with material (1). Turning a part's G-code into a 3D grid of such voxels proceeds in three steps.

First, the G-code is parsed line by line, tracking the nozzle's position (X, Y, Z) and how much filament has been extruded so far. Two commands set how that extruded amount is interpreted: `M83` puts the extruder in relative mode, while M82 puts it in absolute mode; `G92 E0` resets that running total to zero without moving the nozzle. The nozzle's moves are given by `G0` (a non-extruding travel move) and `G1` (a coordinated move, normally extruding). Only `G1` moves that travel in X/Y while extruding are kept, yielding a list of extruded segments.

Second, the bounding box of these segments i.e. the smallest box aligned with the X, Y, Z axes, that contains all of them is computed. It sets the extent of the voxel grid. The grid's resolution is set independently for each axis: the voxel size along Z is set equal to the layer height h so that each printed layer maps to exactly one voxel slice, the voxel size in the XY plane is a separate freely chosen parameter p (the pitch).

Third, each segment is rasterised: every voxel within a radius

$$r = \max\left(1,\ \text{round}\left(\frac{d_{\text{nozzle}}/2}{p}\right)\right) \tag{3.5}$$

of the segment's path (in voxels, within the XY plane) is set to 1 (filled) – d_{nozzle} being the nozzle diameter – giving the rasterised trace a thickness of approximately d_{nozzle}. Repeating this for every segment yields a binary 3D volume that reproduces the geometry as it was actually printed.

3.3.2 From a Voxelised Volume to a Reconstructed Density

This voxelised volume is the input to TomoPy, an open-source CT library that, in this thesis, stands in for a physical X-ray CT scanner. TomoPy expects this input as a 3D NumPy array of shape (Z, Y, X). Two functions carry out the simulated scan: `project(vol, angles)` simulates the projections an X-ray detector would record, one projection per angle in the `angles` array (in radians) and the function `recon(proj, angles, algorithm)` then reconstructs an approximation of the original density volume from those projections, given the same angles and a choice of reconstruction algorithm.

The corners of a reconstructed slice do not correspond to any real projection and accumulate artefacts: values with no physical meaning, far too high or far too low. `circ_mask(recon, axis, ratio)` discards them by zeroing out everything outside a circle of radius `ratio`. For the reconstruction algorithm itself, `gridrec` was retained: a Fourier-based method, it is a fast and a reliable algorithm.

3.3.3 Evaluation Criteria

Table 3.5 maps the four criteria of Section 2.2.2 onto the concrete tests used for this thesis.

Criterion	Test
Capacity	Payload size achievable per method, as a function of the number of layers and lines available
Robustness	Bit-error rate obtained using TomoPy as the simulated scanning process
Fidelity	Physical print inspection
Security	Presence and strength of a secret-key dependency in the bit-to-position mapping

Table 3.5: Mapping of the four watermarking evaluation criteria

3.4 Chapter 3 Summary

This chapter has led to the following decisions, developed throughout the sections indicated:

- **Modulation principle:** every method encodes a bit by tilting or rotating an infill element by a small angle (Section 3.1).
- **Payload:** a 64-bit locally generated opaque identifier is retained (Section 3.2.1).
- **Error handling:** redundancy through repetition (Method 2) or hash collisions (Method 3) combined with majority voting, with a 16-bit CRC checksum used to flag a corrupted copy of the payload (Sections 3.2.2 and 3.2.3).
- **Validation strategy:** decoding is validated by simulating the acquisition pipeline with TomoPy (Section 3.3).

Chapter 4

Implementation

This chapter describes how the decisions taken in Chapter 3 were turned into running code: a modified build of PrusaSlicer that embeds a payload while generating the infill, and a separate Python toolchain that turns the resulting G-code into a simulated CT reconstruction and decodes it back.

Section 4.1 explains how PrusaSlicer's own pipeline works and where each of the three methods hooks into it. Section 4.2 covers the code shared by every method: turning a part's G-code into a voxelised volume and simulating its CT acquisition with TomoPy. Sections 4.3 and 4.4 detail the encoding and decoding code for Methods 1 and 2, respectively. Finally, Section 4.5 states what would still need to be built for Method 3, left unimplemented, before Section 4.6 closes the chapter.

4.1 Integration into PrusaSlicer

Since the watermark is inserted in the infill, the first task was to understand PrusaSlicer's internal pipeline well enough to locate which part of the software actually generates the infill, so that the encoding logic described in Section 3.1 could be hooked in at the right point. This section describes that pipeline and where each method attaches to it.

4.1.1 PrusaSlicer Pipeline

At the highest level, PrusaSlicer takes an STL model as input and produces G-code instructions as output. Internally, this conversion proceeds in three broad stages: (1) the 3D model and the print configuration (layer height, speed, infill density, and so on) are loaded; (2) a printable object data structure is built from them; (3) the configuration is applied and this object is processed (slicing it into layers, then generating its perimeters, infill, top/bottom surfaces, supports, and so on) until the result is ready to be written out as G-code. Figure 4.1 traces the data representation through these stages.

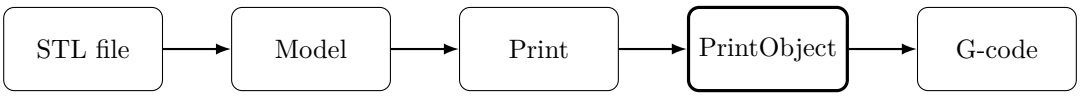


Figure 4.1: Data representations along the slicing pipeline. `PrintObject` is the structure built and progressively filled in during processing, and the one this thesis's encoding logic operates on.

`PrintObject` organises the part being sliced as a hierarchy of three levels. A `Layer` is one horizontal slice of the object: it stores its height, the 2D contour obtained by slicing the model at that height, and a list of regions. A `LayerRegion` groups the geometry sharing the same role and printing parameters within a layer (for example: perimeter, top/bottom surface, internal

(sparse) fill, or support) together with the parameters that apply to it (density, infill pattern, number of perimeters, and so on). Each region in turn holds a list of surfaces: plain 2D polygons, each tagged as top, bottom, or internal, representing the exact area to be filled. The infill is generated from these surfaces.

Infill generation itself is organised around a `Fill` base class, where every concrete infill pattern (`FillRectilinear`, `FillGrid`, `FillHoneycomb`, `FillGyroid`, etc.) is a subclass overriding a virtual method that generates the actual polylines for one region of one layer given a set of shared parameters (angle, spacing, layer index, and so on). A factory function, `Fill::new_from_type()`, maps an infill-pattern enum value to the corresponding subclass instance. Per printed layer, `Layer::make_fills()` (in `Fill.cpp`) is invoked once: it iterates over the regions of that layer requiring infill, instantiates the appropriate `Fill` object through the factory, sets the shared parameters, and triggers geometry generation. Critically, PrusaSlicer processes layers concurrently (multi-threaded), which shaped both methods' designs.

4.1.2 Insertion of the Methods

Here is how the three methods are inserted into PrusaSlicer:

- **Method 1** hooks in immediately after the generic `Fill` object is created and before any pattern-specific dispatch: if the current layer index falls within the watermark's layer range, the shared `angle` parameter is perturbed by α before being handed to whichever pattern the user selected (Section 4.3). Because this only touches the generic, Method 1 works with any infill pattern PrusaSlicer supports.
- **Method 2** is registered as an entirely new infill pattern (`FillWatermarkGrid`), selectable through PrusaSlicer's Graphical User Interface (GUI). Inside `FillWatermarkGrid`, each candidate line's position in the bit stream is computed as a pure function of the layer index and its index within its family specifically to remain correct regardless of the order in which layers are actually processed.
- **Method 3** is inserted in exactly the same way as Method 2.

Decoding operates on a CT reconstruction obtained after slicing, printing and recovery of a physical part, and is implemented entirely outside PrusaSlicer.

4.2 Validation: From G-code to a Reconstructed Density

Section 3.3 already explains, at a conceptual level, why decoding is validated through a simulated acquisition rather than a real CT scan, and what the two stages of that simulation do. This section shows that pipeline actually working.

4.2.1 G-code Parsing and Voxelisation

`gcode_to_voxels.py` implements the G-code parsing and rasterisation logic already described in Section 3.3 (tracking nozzle position and extrusion state, computing a bounding box, then rasterising each extruded segment); the full `parse_gcode()` function is listed in Appendix A (Listing A.1).

An issue surfaced only once that description was actually implemented. PrusaSlicer's G-code output begins with a priming sequence before the object itself; these moves also extrude material while moving in X/Y, and would otherwise be picked up as spurious segments. The parser instead waits for the first `;LAYER_CHANGE` comment, which PrusaSlicer emits right before the object's first real layer, and ignores every line before it.

Figure 4.4 validates the result: layer 5 of a watermarked test cube's grid infill, as rendered by PrusaSlicer's own layer preview, against the same layer after parsing and voxelisation, using `pitch` = 0.2 mm, `nozzle_diameter` = 0.4 mm and `layer_height` = 0.2 mm. The two agree on the geometry of the infill, confirming that the parser correctly reproduces what was actually sliced.

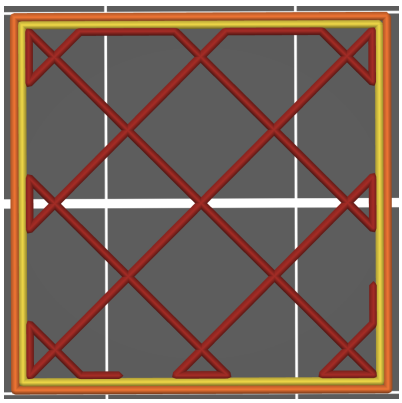


Figure 4.2: Layer 5 in PrusaSlicer

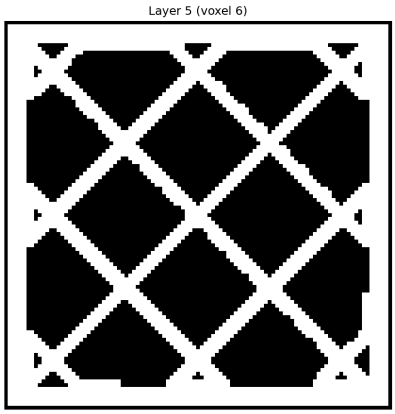


Figure 4.3: Layer 5 voxelised

Figure 4.4: Layer 5 of a watermarked 2 cm test cube's infill: PrusaSlicer's own layer preview (left) against the same layer after G-code parsing and voxelisation (right)

4.2.2 Simulated CT Acquisition

Section 3.3 already explains what `project()`, `recon()`, `circ_mask()` and `gridrec` do. Before applying this pipeline to an actual watermarked part, it was first exercised on `shepp3d()`, a test volume bundled with TomoPy itself, as a sanity check that the installed pipeline behaved as expected (Listing A.2). Figure 4.5 shows the resulting voxelised volume, sinogram, and reconstruction at $Z = 32$, the middle of the object. A sinogram is simply the image formed by the projections at one fixed Z height

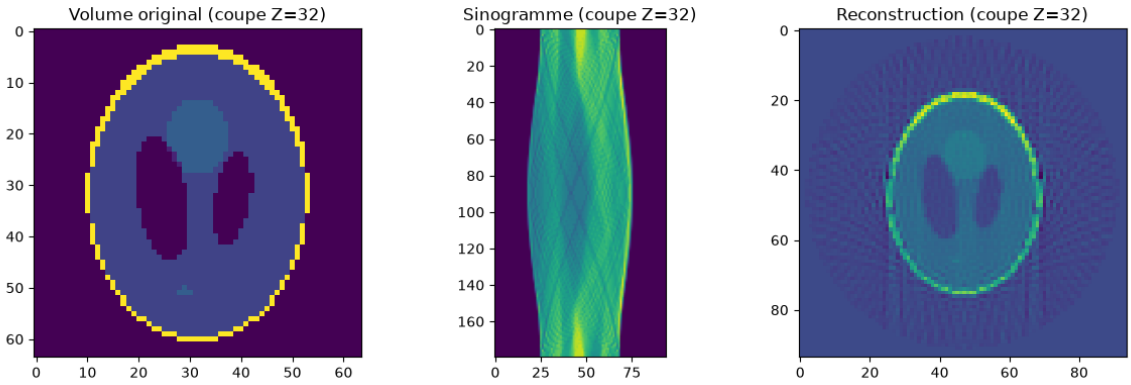


Figure 4.5: Voxelised volume, sinogram, and reconstruction for TomoPy's built-in `shepp3d()` test volume, at $Z = 32$.

The same pipeline was then applied to the voxelised volume of Section 4.2.1. Figure 4.6 shows the result on layer 5 of the same cube as Figure 4.4: the reconstruction preserves the infill pattern well enough for the decoding steps.

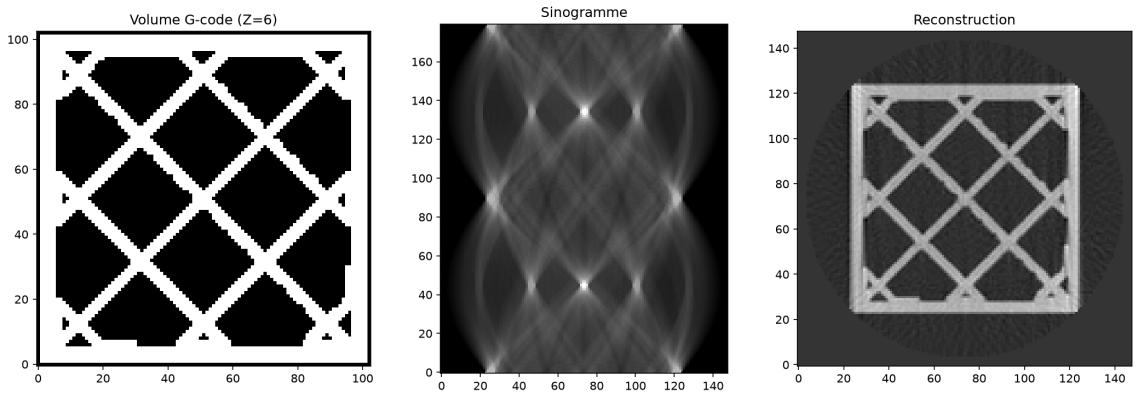


Figure 4.6: Voxelised volume, sinogram, and TomoPy reconstruction of layer 5 of a watermarked test cube (compare with the reference layer of Figure 4.4)

4.3 Method 1: Whole-Layer Orientation Modulation

4.3.1 Encoding

Method 1 requires no new infill pattern: it reuses whichever pattern the user already selected and simply adds a fixed offset to that layer's infill angle. Listing A.3 shows the relevant block, inserted directly in `Layer::make_fills()` where every layer's `Fill` object is configured.

To generate the opaque identifier, a function draws a random 64-bit sequence (see Listing A.4). The `WatermarkData` structure shown in Listing A.5 then uses this identifier, inside `Layer::make_fills()`, to build the watermark payload. As shown there, this relies on a function which concatenates the randomly generated identifier with the checksum computed by `compute_watermark_checksum()` (see Listing A.6).

In Figure 4.9, the infill's orientation can be seen to have changed. This is most noticeable at the contact points between the perimeter and the infill, where the shape of the infill is visibly different at those locations (see the blue box in Figure 4.9).

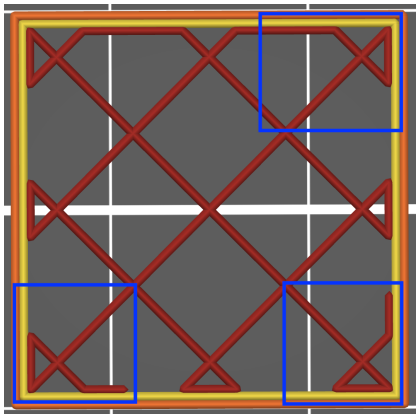


Figure 4.7: Layer 5 in PrusaSlicer

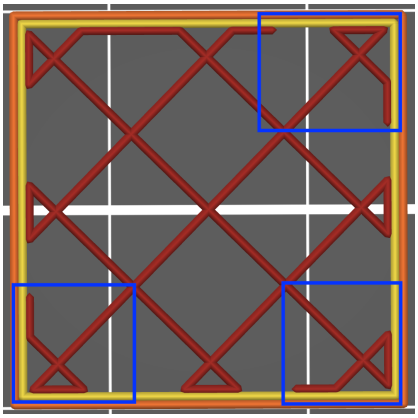


Figure 4.8: Layer 6 in PrusaSlicer

Figure 4.9: Layers 5 (left) and 6 (right) of a watermarked 2 cm test cube's infill in PrusaSlicer's own layer preview

It is precisely because of these contact points between the perimeter and the infill that this method's fidelity is lower than that of the other methods (Section 5.2.1)

4.3.2 Decoding

Listing A.7 shows the decoding logic implemented in `decoding_method1.py`. `mean_matrix` first collects, for every reconstructed layer, the mean difference between that layer and the previous one: a low value indicates that the current and previous layers carry the same bit, while a high value indicates that they carry different bits. Each value in `mean_matrix` is then compared against a fixed threshold of 10^{-6}, chosen empirically: as Figure 4.10 shows, the low-difference population sits around 10^{-8} and the high-difference population around 10^{-4} (for a 2 cm side cube).

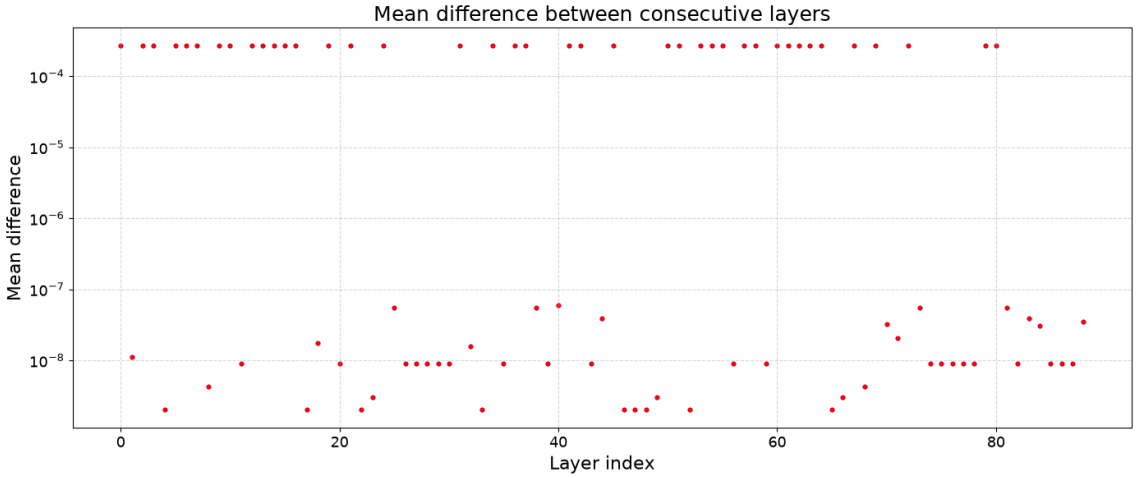


Figure 4.10: Mean difference between consecutive reconstructed layers, for a 2 cm cube watermarked with Method 1

This yields a single, continuous bit stream spanning every decoded infill layer, which is then split into consecutive 80-bit copies of the payload. Each copy is checked against its own checksum with `verify_watermark_checksum()` (Listing A.8), and any copy that fails this check is discarded before voting, following the strategy described in Section 3.2.3. The remaining, validated copies are then combined bit by bit through a majority vote to produce the final 80-bit payload. The script finally compares this payload against the known test payload; for the test cube used throughout this chapter, the two sequences matched exactly.

4.4 Method 2: Per-Line Grid Modulation

Unlike Method 1, Method 2 cannot reuse an existing infill pattern, since it needs to control the orientation of individual grid lines rather than a whole layer.

4.4.1 Encoding

The second encoding method builds on PrusaSlicer's standard grid infill pattern. The grid is generated as two families of parallel lines, one vertical and one horizontal, and each individual line is used to carry a single bit of the payload: the line is left straight to encode a '0', or rotated by a small fixed angle around its center to encode a '1' (Listing A.10). For every layer above the solid bottom layers, the mapping between grid lines and payload bits is computed from the layer index and the line's position within its family, and this mapping wraps around the payload so that the watermark is repeated redundantly across the height of the print (Listing A.9). As in Method 1 (4.3.1), the watermark payload itself is generated in the same way, i.e. converted into a binary vector prior to slicing.

Figure 4.11 shows the result of Method 2 with an exaggerated tilt angle (4°) to clearly illustrate the effect of the encoding, using the payload $[1, 1, 0, 1, 1, 0, 1, 0]$, as shown schematically in Figure 3.2.

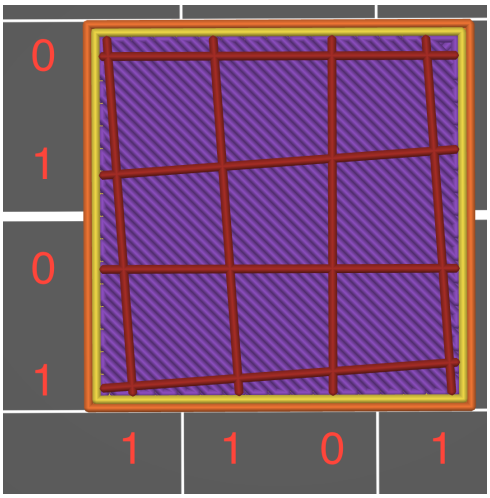


Figure 4.11: Grid infill of layer 5 encoded with Method 2 using an exaggerated tilt angle of 4°

4.4.2 Decoding

Decoding, implemented in `decoding_method2.py`, works on the `recon` array of Section 4.2.2 and starts by zeroing every reconstructed value below 0.35 to suppress reconstruction noise (Listing A.11). Decoding relies on the pair of calibration layers introduced in Section 3.1.2: the first infill layer, forced to bit '0' on every line, and the second, forced to bit '1'. Their absolute difference is summed column-wise and row-wise into `col_peak_ref` and `line_peak_ref`, two 1D profiles with a peak wherever a line sits, since every line differs between an untilted and a fully tilted layer while the rest of the volume cancels out.

Each reference peak's index range is located by `detect_peaks_reference()` (Listing A.12): applied to `col_peak_ref` and `line_peak_ref`, this yields `regions_col` and `regions_line`: lists of tuples containing the starting index and the ending index of a peak.

For every subsequent layer, the same column-wise and row-wise summation is applied to the absolute difference between that layer and the untilted calibration layer, producing `col_peaks` and `line_peaks`. `compare_peaks()` (Listing A.13): it compares the mean value of the reference profile and of the current layer's profile over that region, and reports a '1' if the relative gap between the two means stays within `gap_threshold` and a '0' otherwise. Repeating this layer by layer, reconstructing the full watermark bitstream to be compared against the known payload, exactly as in Method 1 (Section 4.3.2).

Figure 4.12 illustrates this on layer $Z = 8$ of a watermarked test cube, whose four columns on that layer encode $[1, 1, 0, 1]$: the reference profile (top) shows four comparable peaks, while layer 8's profile (bottom) keeps the reference's amplitude on the first, second and fourth peaks and collapses to a small residual on the third – the $[1, 1, 0, 1]$ pattern, correctly read off by `compare_peaks()`.

`decoding_method2.py` also embeds a small helper, which dumps a given matrix to a colour-scaled spreadsheet; this was used during development to inspect `reference` and `abs_diff_matrix` directly, rather than through their column/row sums. Figure 4.13 shows both exports for the same cube: `reference` (left) is red along every printed line, confirming that the calibration difference used to build `col_peak_ref`/`line_peak_ref` covers the whole grid, while `abs_diff_matrix` for layer $Z = 8$ (right) is red only on the lines carrying a '1' bit, matching Figure 4.12. Both

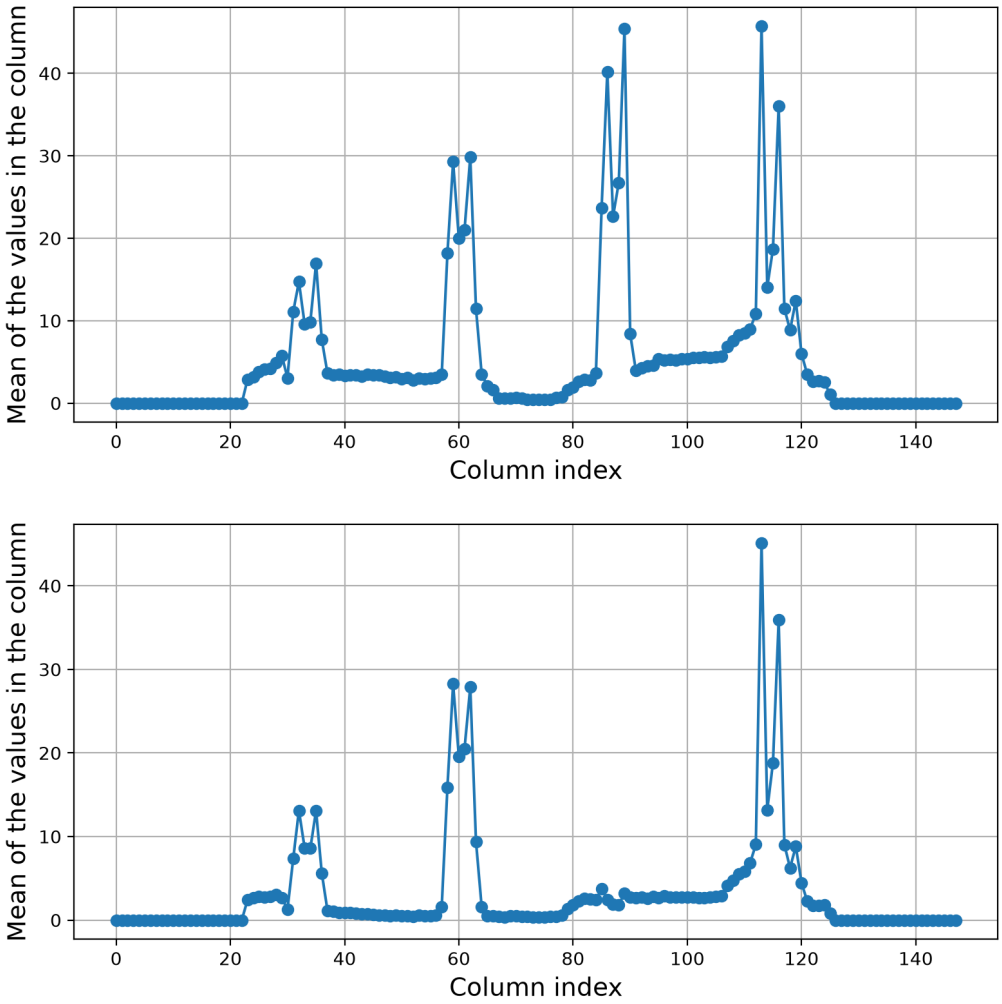


Figure 4.12: Column peak reference `col_peak_ref` (top) against the column profile `col_peaks` of layer $Z = 8$ (bottom)

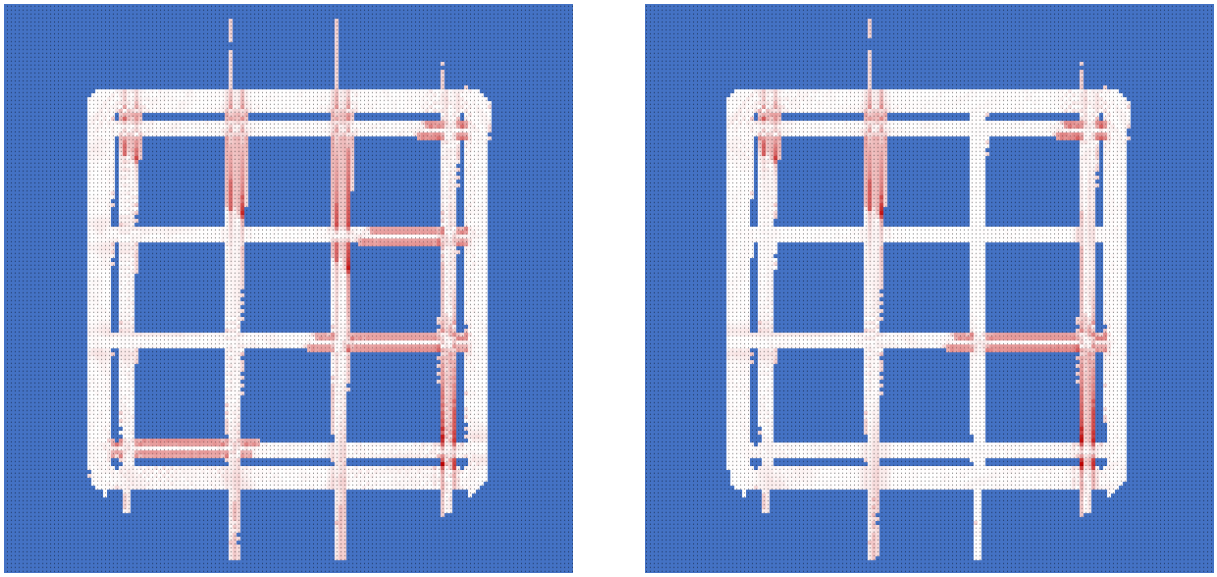


Figure 4.13: Colour-scaled export of `reference` (left) and of `abs_diff_matrix` for layer $Z = 8$ (right): the reference differs along every line, while layer 8 differs only along the lines encoding a '1' bit.

exports also show that the 0.35 threshold applied at the start of decoding does not remove every reconstruction artefact: isolated red pixels remain outside the printed grid in both matrices. They do not appear to affect decoding here, since `compare_peaks()` works on a region's mean rather than on individual voxels, but they are a reminder that this filtering step is not a complete noise removal and may need to be revisited once a full test set is available.

4.5 Method 3: Hash-Based Line Indexing

Unlike Methods 1 and 2, Method 3 exists only as the design and worked numerical example of Section 3.1.3. This section states what would need to be built to close that gap.

- On the encoding side, Method 3 would reuse `FillWatermarkGrid`'s per-line geometry unchanged (Section 4.4.1); the only change is how a line's bit-stream position is obtained. Instead of the pure function of layer index and family index used by Method 2, each candidate line's position (x, y, z) would be passed through a keyed hash $h(x, y, z, K)$ (Section 3.1.3) to obtain $(idx, flip)$.
- On the decoding side, the same hash would need to be reimplemented and applied to every detected line's position after reconstruction, to recompute $(idx, flip)$ and accumulate the majority vote for each payload index described in Section 3.1.3.

4.6 Chapter 4 Summary

This chapter has described the concrete implementation underlying the results discussed in Chapter 5:

- **Validation pipeline:** `gcode_to_voxels.py` reconstructs a voxelised volume from a printed part's toolpath before this volume is projected and reconstructed with TomoPy (Section 4.2).
- **Method 1:** `decoding_method1.py` decodes it via consecutive-layer differencing (Section 4.3).
- **Method 2:** `decoding_method2.py` decodes it via column/line difference profiles and peak detection (Section 4.4).
- **Method 3:** left unimplemented on both the encoding and decoding side (Section 4.5).

Chapter 5

Results and Discussion

This chapter reports and discusses the results obtained by exercising Methods 1 and 2 (Sections 3.1.1, 3.1.2, 4.3 and 4.4) across the experimental protocol of Section 5.1, evaluates the robustness and fidelity and discusses the capacity and security introduced in Section 3.3.3.

Section 5.1 describes the tests used to evaluate the robustness of Methods 1 and 2 and presents the results for each. Section 5.2 presents the fidelity of Methods 1 and 2, showing photos of printed test prints and assessing the quality of the watermark. Section 5.3 discusses what these results mean for each method and what could be improved.

5.1 Robustness

Robustness is tested along two axes: the cube's side length $c \in \{1, 2, 3\}$ cm, and the target displacement $d \in \{60, 70, 80, 90, 100, 110, 120, 130, 140\}\,\mu$m used to size the tilt/rotation angle (Equation (3.1)). Table B.1 reports the tilt angles obtained from this equation as a function of the cube size and the displacement.

For every configuration, decoding is run on the simulated CT reconstruction of Section 4.2.2 and produces a bit stream, which is compared against the known test payload bit by bit. Three numbers are reported per configuration:

- The number of complete copies detected.
- The checksum outcome: whether the 16-bit CRC of Section 3.2.2 accepts or rejects the assembled payload.
- The Bit Error Rate (BER): the fraction of assembled payload bits that differ from the known test payload;

5.1.1 Method 1: Whole-Layer Orientation Modulation

Table B.2 reports, for every (cube size, displacement) pair, the number of complete payload copies the decoder was able to isolate. Table B.2 shows that the 1 cm cube does not contain enough layers to hold a single copy of the payload, while the 2 cm and 3 cm cubes hold only one copy each, regardless of displacement. Because Method 1 encodes one bit per layer (Section 3.1.1), its capacity is fixed by the layer count alone and, unlike Method 2, does not grow with in-plane cube size, only with part height. With a single copy available, no majority vote is possible for the two larger cubes, so the robustness of Method 1 rests entirely on the reliability of each individual layer read.

Table B.3 reports the percentage of the single detected copy whose 16-bit CRC accepts, and Table B.4 reports the corresponding BER. For the 2 cm and 3 cm cubes, decoding is perfect

across the whole range tested (0 % BER and 100 % checksum pass rate), with no exceptions. Even the smallest displacement tested, 60 μm, already decodes without error.

This near-perfect robustness is explained by the very large gap between the mean voxel intensity of two consecutive layers, visible in Figure 4.10. Method 1 changes the orientation of the infill for an entire layer at once, which displaces every contact point between the infill and the perimeter (as shown in Figure 4.9). Each of these contact points locally alters the deposited material, so summing the effect over an entire layer produces a shift in mean intensity.

This large contrast between two layers is what makes Method 1 so robust, but it comes at the cost of a degraded fidelity: changing the infill orientation of a whole layer is also what makes the watermark potentially visible or otherwise detectable. Fixing this would require designing a custom infill grid. However, since Method 1's capacity is fixed by the number of layers, this low capacity was the deciding factor in prioritising the implementation effort on Method 2 instead.

5.1.2 Method 2: Per-Line Grid Modulation

Table B.5 reports, for every (cube size, displacement) pair, the number of complete payload copies the decoder was able to isolate before the majority vote (see Figure 5.1).

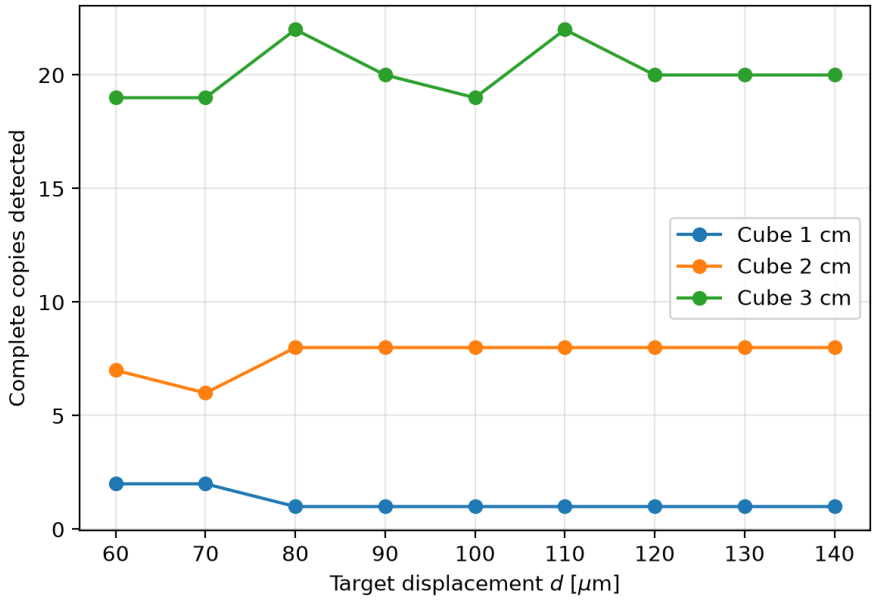


Figure 5.1: Number of complete copies detected as a function of the target displacement, by cube size

We can see that the number of copies scales with cube size (1–2 for the 1 cm cube, 6–8 for the 2 cm cube, 19–22 for the 3 cm cube) and is essentially flat across displacement for a given size. This is consistent with the theoretical capacity of Section 3.1.2, $(N_x + N_y)\times$ number of layers. The residual variation at fixed size (e.g. 7 and 6 copies at 60 and 70 μm for the 2 cm cube, against a stable 8 from 80 μm onward) is attributed to the reference-line peaks used to delimit each family being harder to localise precisely at small displacements.

Since the 1 cm cubes at 60–70 μm displacement only contain two complete copies, ties between bits are frequent, which leads to a high BER (see Table B.8). The same happens for the 3 cm cube.

Table B.7 reports, for each configuration, the percentage of the individually detected copies whose own 16-bit CRC accepts before any majority vote is applied.

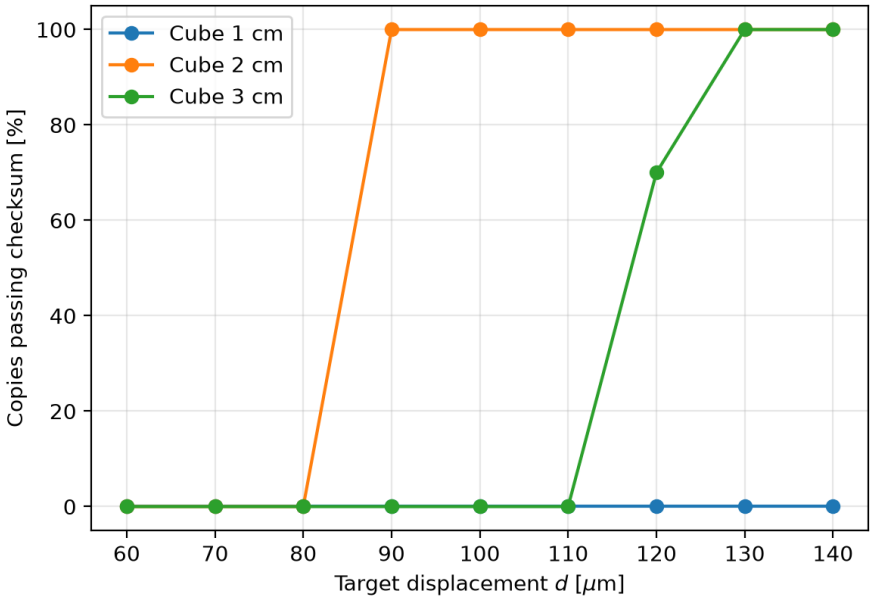


Figure 5.2: Percentage of individually detected copies passing their own CRC, as a function of the target displacement, by cube size

Figure 5.2 shows that checksum acceptance is essentially binary: it sits at 0 % until a displacement threshold, then jumps to 100 %, and never leaves 0 %. The 16-bit CRC is far stricter than the raw BER might suggest: the 1 cm cube's BER drops to 6–11 % from 80 μm onward (Table B.8), i.e. roughly 90 % of bits are already correct, yet not a single copy ever passes its checksum.

Table B.8 reports the final BER, computed after the redundancy/majority-vote step, against the known test payload.

Figure 5.3 shows that below 70–80 μm threshold, all three cube sizes have 40 % to approximately 50 % of BER. Above that threshold, the three curves separate:

- The 1 cm cube suddenly improves but then stays around 6–11 % instead of reaching zero: with only one non-redundant copy from 80 μm onward, there is no majority vote left to correct the errors.
- The 2 cm cube recovers cleanly, falling from 50 % to 11.25 % between 70 and 80 μm and reaching a stable 0 % from 90 μm onward, matching the point where its checksum acceptance also reaches 100 %.
- The 3 cm cube is the least monotonic of the three: BER stays high and noisy through 80–110 μm (27.5–53.75 %) despite having 19–22 copies available, before dropping to 0 % from 120 μm onward. Note that at 120 μm the majority vote already reaches a perfect 0 % BER even though only 70 % of the individual copies pass their own checksum (Table B.7): with 20 copies available, the vote is strong enough to outvote a minority of corrupted copies.

Taken together, Tables B.5 to B.8 point to three conclusions for Method 2. First, checksum-validated decoding is only achieved for the 2 cm and 3 cm cubes; the 1 cm cube never validates over the range tested, because its capacity is too small to hold more than one or two copies of the payload. Second, once a size/displacement pair reaches its BER floor, further increasing d brings no additional benefit. Third, the 16-bit CRC used as the accept/reject criterion (Table B.7) is considerably stricter than the BER alone suggests.

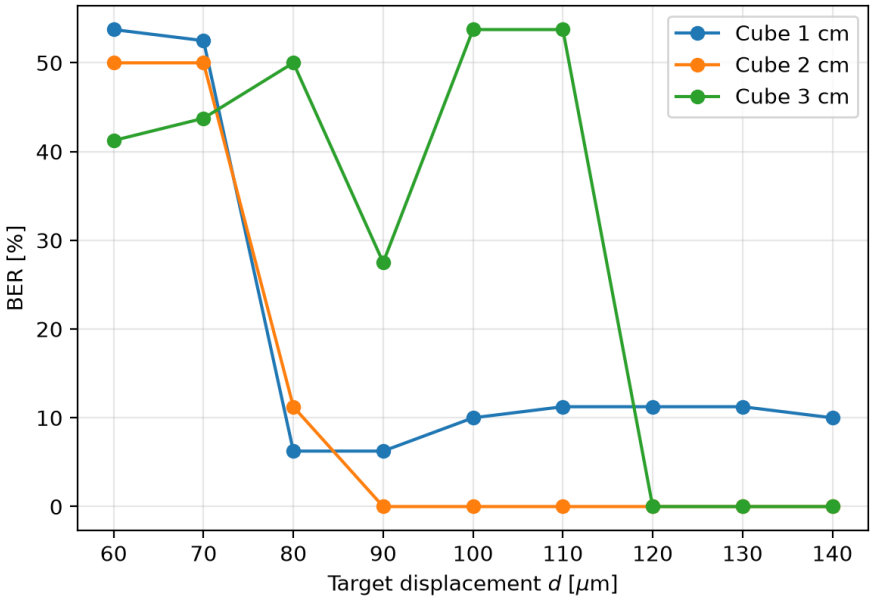


Figure 5.3: Final BER (post majority vote) as a function of the target displacement, by cube size

5.2 Fidelity

To assess the fidelity of each method, prints were produced on the Prusa MK4S at the Makilab (the same printer used for the calibration tests of Table 3.1).

5.2.1 Method 1: Whole-Layer Orientation Modulation

For Method 1, 1, 2, and 3 cm cubes were printed at three displacements (60, 100, and 140 μm) and stopped at roughly 30 % of their height to expose the infill. Figure 5.4 shows the resulting nine partial prints.

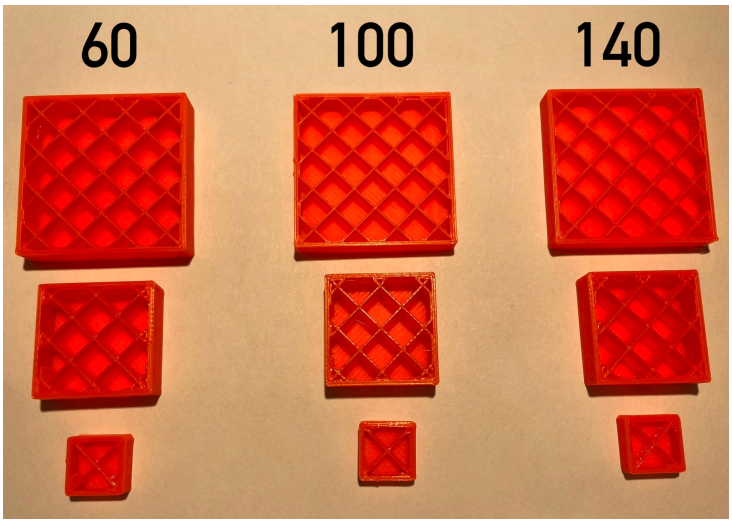


Figure 5.4: 1, 2, and 3 cm cubes printed with Method 1 at 60, 100, and 140 μm displacement (partial prints)

At first glance, none of the nine parts betrays that it carries a watermark. At 60 μm the infill looks like an ordinary print, whereas at 140 μm the wall formed by the stacked infill layers is visibly rougher, with the boundary between consecutive layers standing out more sharply.

Figure 5.5 contrasts the two displacements on the 3 cm cube.

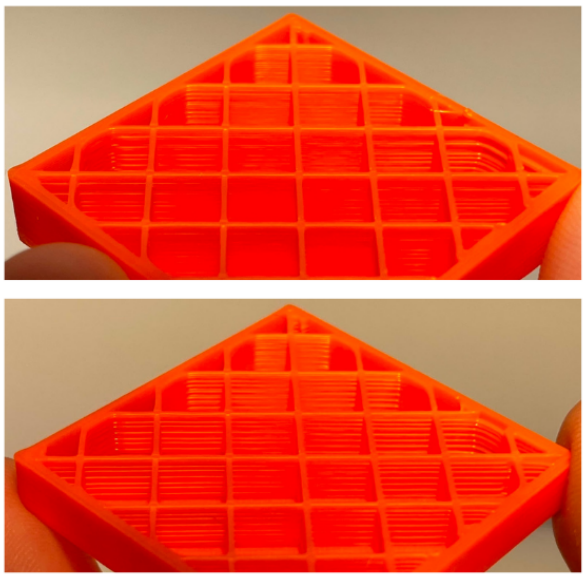


Figure 5.5: Infill layers for the 3 cm cube at 60 μm (above) versus 140 μm (below) displacement

The roughness traces back to the perimeter/infill contact points discussed in Sections 4.3.1 and 5.1.1: each is a small, localised distortion, but stacked over many layers they read as a visible alternation. Figure 5.6 zooms in on this pattern for the 2 cm cube at 100 μm (the choice of cube is arbitrary, since the same alternation appears at this contact point on every cube printed, regardless of size or displacement). Detecting it, however, requires deliberately looking for it at that specific location; it is not something a casual observer would notice.

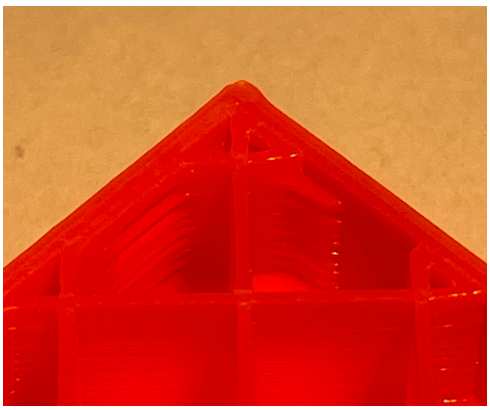


Figure 5.6: Alternation visible at the perimeter/infill contact point

5.2.2 Method 2: Per-Line Grid Modulation

Figure 5.7 shows the three cubes printed with Method 2 at a 140 μm displacement, again stopped partway through. Unlike Method 1, these prints show a clear defect: on every cube, a visible gap separates the infill from the perimeter (Figure 5.8, a close-up of the 1 cm cube). This contradicts the PrusaSlicer preview, where the infill lines meet the perimeter as expected (Figure 4.11), and the cause remains unidentified: extending the line length at creation time in `build_family()` (Listing A.10) did not close the gap. Resolving this difference between the simulated and the printed geometry is left for future work.

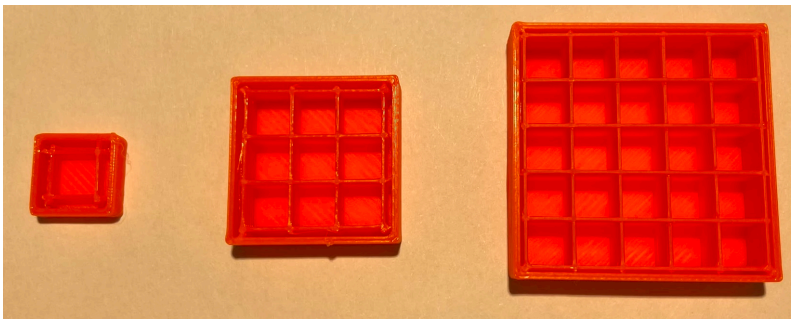


Figure 5.7: 3 partially printed cubes for Method 2 at 140 μm displacement

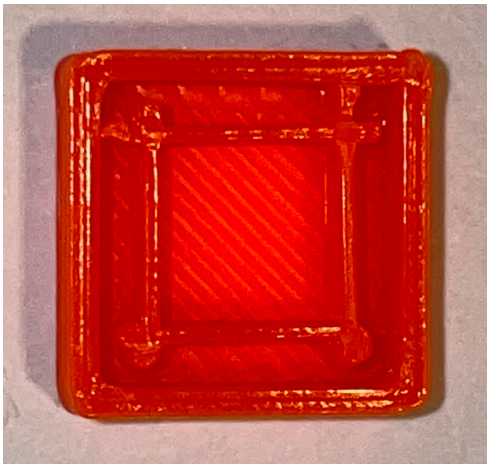


Figure 5.8: Close-up of the 1 cm cube showing the gap between infill and perimeter

This gap is a direct fidelity issue on its own, but printing also surfaced a second one that the simulations had not flagged: some infill lines run extremely close to the cube's edge, which both parasitises decoding and looks suspicious to an outside observer. This may well be why PrusaSlicer's built-in grid pattern is oriented at 45° rather than axis-aligned. Here the grid had deliberately been kept axis-aligned instead, purely to simplify decoding. Beyond these two issues, the general state of the infill matches the conclusion reached for Method 1.

A more practical observation from the printing sessions themselves: the nozzle is large enough to block the view of the infill while a layer is being deposited (depending on the size of the object compared to the nozzle). Section 2.3.3 argued for keeping the modulation small partly on the assumption that someone could watch the infill build up in real time; since the nozzle itself limits that visibility, a larger modulation could plausibly be tolerated without being noticed during printing. That said, a failed print exposes the infill afterwards regardless, so keeping the modulation small remains the safer choice.

5.3 Discussion

5.3.1 Method 1

Because encoding only perturbs the shared angle parameter handed to whichever pattern the user selected (Section 4.1), it works with any infill pattern, unlike Methods 2 and 3, which both require the dedicated `FillWatermarkGrid` pattern (Table 3.3). Method 1 reaches a 0 % BER and a 100 % checksum pass rate across the whole 60–140 μm range tested, for both the 2 cm and 3 cm cubes (Section 5.1.1).

This robustness needs to be qualified, however. Table B.2 shows that the 2 cm and 3 cm

cubes each hold exactly one copy of the payload, and the 1 cm cube none at all: reaching the 241 infill layers needed for a majority vote would require a considerably taller print than any tested here. The perimeter/infill contact points across the whole layer (Figure 4.9) are also what limits fidelity: Section 5.2.1 found the infill wall growing visibly rougher as displacement increases, with a clear alternation at these contact points once one knows to look for it. Method 1 also shares two limitations with Method 2: its bit-to-layer mapping assumes a constant cross-section, and it offers no security, since this mapping is fixed and public.

Two improvements follow directly. First, the offset angle α is currently hard-coded for cubes rather than recomputed from Equation (3.1) using the object's actual infill geometry. Second, since the visibility issue traces back to the perimeter contact points rather than to the rotation itself, generating a custom infill grid designed to suppress these contact points (as in Method 2) could improve the fidelity.

5.3.2 Method 2

Method 2 answers the capacity limitation identified for Method 1. Table B.5 finds 6–8 complete 80-bit copies for the 2 cm cube and 19–22 for the 3 cm cube, against a single copy for Method 1 at the same sizes (Table B.2). Figure 5.3 shows redundancy actually correcting isolated errors.

This capacity comes at a cost, however. Method 2 requires a dedicated custom infill (`FillWatermarkGrid`), so a user has to deliberately select this grid pattern to obtain a watermarked part, unlike Method 1; extending the approach to arbitrary infill patterns is left as future work. The detection of the reference peaks used to delimit each line is the main source of the errors that persist (Section 5.1.2) – somewhat surprisingly, the peak marking the first line of family X is consistently weaker than the others and sometimes goes undetected.

The validation reported throughout this chapter also relies entirely on TomoPy standing in for a real CT scanner. In practice, reconstruction quality on a physical scanner is more likely to be worse than better. Printing itself introduced issues specific to this method (a gap between infill and perimeter that extending line length in `build_family()` did not close, and grid lines running close to the cube's edge (Section 5.2.2)) which bias the fidelity measured here. Like Method 1, Method 2 offers no security and assumes a constant cross-section.

Another limitation is that Method 2 implicitly assumes the fragment's orientation is already known (how the object is rotated about the vertical axis) which cannot be assumed once a part is recovered without its original file. One option worth exploring is to leave a distinguishable mark in the solid top or bottom layers that could serve as an orientation reference during scanning, so the fragment can be correctly aligned before the per-layer or per-line decoding logic itself is applied.

Three improvements follow. Adding key-dependent bit positioning (Method 3, Section 3.1.3) would close the security gap shared with Method 1. Improving reference-peak detection, specifically for the first line of each family, would likely reduce the residual error rate further. Finally, resolving the infill/perimeter gap surfaced during fidelity testing, most likely within `FillWatermarkGrid`'s line-generation logic, would improve fidelity.

5.3.3 General Discussion

Methods 1 and 2 trade the same two criteria but in opposite directions: Method 1 sacrifices capacity for an excellent robustness and unconditional compatibility with any infill pattern, while Method 2 sacrifices this robustness for a good capacity leading to a good redundancy. Neither comes close to satisfying every criterion of Table 3.5 at once, and both share the same three limitations discussed below.

The most significant of these is the absence of security: both methods rely on a bit-to-position mapping that is fixed and public once the encoding scheme itself is known. Method 3's key-dependent hashing (Section 3.1.3) is the natural answer to this gap. This method being left unimplemented (Section 4.5) is the most significant item this thesis leaves open.

A second limitation is that all robustness figures are based on a TomoPy-simulated CT acquisition standing as a physical scanner. This substitution reproduces the reconstruction step of a CT scan, but not necessarily its potential noise. The BER values reported for both methods should therefore be read as evidence that the encoding schemes are decodable in principle from a reconstructed density volume, not as a prediction of what a real scanner would recover.

Finally, every result in this thesis was obtained on PLA cubes printed on a single Prusa MK4S. This is a deliberate choice: FDM/PLA was retained because it is both the combination made available by the Makilab and, per existing surveys, representative of how illegally printed firearm components are actually manufactured (Section 2.3.2). Extending validation to other materials, printers and non-constant cross-sections remains necessary before the conclusions drawn here can be generalised to the full range of parts this watermark would ultimately need to cover.

Chapter 6

Conclusion

This thesis addresses a problem specific to the growing accessibility of 3D printers: when someone prints a component of a firearm and it is found, the object itself is the only clue to identifying who printed it (Chapter 1). The approach used in this thesis was to encode an identifier into the infill during the slicing process in a way that is imperceptible.

To achieve this aim, we had to make well-founded choices as outlined in Chapter 2: blind detection, integration into the slicing software rather than the mesh or G-Code, FDM/PLA on a Makilab Prusa MK4S printer, and an opaque identifier serving as the payload.

Chapters 3 and 4 translate these choices into three encoding processes: Method 1, which involves tilting the orientation of the infill layers; Method 2, which tilts the lines of the infill grid; and Method 3, a theoretical approach, which extends Method 2 by securing it with a key-dependent hash to determine the encoded bit on the line. These chapters also describe and implement the decoding pipeline using simulated CT reconstruction, and outline the design of a payload containing a checksum to protect the watermark from errors.

In Chapter 5, Methods 1 and 2 were tested and evaluated against the four criteria set out in Table 3.5. Method 1 achieved a BER of 0 % and a checksum pass rate of 100 % across all tests conducted on 2 and 3 cm cubes with displacements ranging from 60 to 140 μm. Although this method is robust, its capacity is limited by the number of infill layers. Method 2 overcomes this capacity limitation by being able to hold between 6 and 8 copies for a 2 cm cube and between 19 and 22 copies for a 3 cm cube, compared with just one for Method 1. The trade-off for this high capacity is that the decoding chain is more fragile. Furthermore, during printing, a gap issue arose between the infill and the perimeter, reducing the accuracy of this method. It is also necessary to know the orientation of the piece in order to carry out the decoding. None of these methods meets all the evaluation criteria, and none offers certainty. This is why Method 3 was devised but has not yet been implemented.

These results show that a payload can be encoded into an FDM/PLA infill during slicing and decoded from a reconstructed density volume without being detected by an average observer. This thesis was unable to demonstrate that the watermark could survive scanning using a real CT scanner. Further work remains to be done, such as: implementing Method 3, validating the decoding using an actual CT scan, resolving the gap issue between the infill and the perimeter for Method 2, adding a reference to determine the original orientation of the printed object, and extending the validation to other materials, other printers and objects with varying cross-sections.

One final point regarding the payload design (Section 2.3.5): the identifier used here can only be used to identify a person if they have a Prusa Connect account, or by comparing the payload with the identifier stored locally in their slicing software

Bibliography

[1] Sébastien Binard and Lawrence Philips. "Watermarking for Additive Manufacturing Copy Control by Roughness Modification". MA thesis. Université catholique de Louvain, 2018. URL: http://hdl.handle.net/2078.1/thesis:14620.

[2] Verified Market Research. *FDM 3D Printer Market Report: Size, Growth, Trends & Forecast (2025–2033)*. 2026. URL: https://www.verifiedmarketresearch.com/product/fdm-3d-printer-market/.

[3] Aja Isaac. *1 Million 3D Printers Sold in Q1 2025: The Brutal Truth About Where Real Profit Hides*. 2025. URL: https://www.shelftrend.com/business-industrial/3d-printer-market-analysis-profit-guide-online-sellers-2025.

[4] Benoît Macq, Patrice Rondao Alface, and Mireia Montanola. "Applicability of watermarking for intellectual property rights protection in a 3D printing scenario". In: *Proceedings of the 20th International Conference on 3D Web Technology*. 2015, pp. 89–95.

[5] Stefan Schaufelbühl, Nicolas Florquin, Denis Werner, and Olivier Delémont. "The emergence of 3D-printed firearms: An analysis of media and law enforcement reports". In: *Forensic science international: synergy* 8 (2024), p. 100464.

[6] Mylène S Falardeau, Caroline Mireault, Benoît Daoust, and Cyril Muehlethaler. "Chemical analysis of polymers used for 3D printing of firearms". In: *Forensic science international* 357 (2024), p. 111999.

[7] Prusa Research. *PrusaSlicer*. URL: https://www.prusa3d.com/fr/p/prusaslicer/.

[8] International Organization for Standardization and ASTM International. *ISO/ASTM 52900:2021 – Fabrication additive — Principes généraux — Fondamentaux et vocabulaire*. 2021. URL: https://www.iso.org/fr/standard/74514.html.

[9] Abdullah Alfaify, Mustafa Saleh, Fawaz M Abdullah, and Abdulrahman M Al-Ahmari. "Design for additive manufacturing: A systematic review". In: *Sustainability* 12.19 (2020), p. 7936.

[10] Y Ji, M Kim, and BJ Lee. "Mass Customization Through Additive Manufacturing: Review of Technologies, Applications, and Scalability Challenges". In: *International Journal of Precision Engineering and Manufacturing-Green Technology* (2026), pp. 1–28.

[11] Simon Ford and Mélanie Despeisse. "Additive manufacturing and sustainability: an exploratory study of the advantages and challenges". In: *Journal of cleaner Production* 137 (2016), pp. 1573–1587.

[12] Laurent Delannay and Aude Simar. *Fabrication mécanique*. Cours LMECA1451, UCLouvain. 2024. URL: https://uclouvain.be/cours-2025-lmeca1451.

[13] Aude Simar. *Advanced manufacturing technologies*. Cours LMECA2453, UCLouvain. 2025. URL: https://uclouvain.be/en-cours-2026-lmeca2453.

[14] Frank Y Shih. *Digital watermarking and steganography: fundamentals and techniques*. CRC press, 2017.

[15] Patrice Rondao Alface and Benoît Macq. "From 3D mesh data hiding to 3D shape blind and robust watermarking: A survey". In: *Transactions on data hiding and multimedia security II*. Springer, 2007, pp. 91–115.

[16] François Cayre and Benoît Macq. "Data hiding on 3-D triangle meshes". In: *IEEE Transactions on signal Processing* 51.4 (2003), pp. 939–949.

[17] Jong-Uk Hou, Do-Gon Kim, and Heung-Kyu Lee. "Blind 3D mesh watermarking for 3D printed model by analyzing layering artifact". In: *IEEE Transactions on Information Forensics and Security* 12.11 (2017), pp. 2712–2725.

[18] Arnaud Delmotte, Kenichiro Tanaka, Hiroyuki Kubo, Takuya Funatomi, and Yasuhiro Mukaigawa. "Blind 3D-printing watermarking using moment alignment and surface norm distribution". In: *IEEE Transactions on Multimedia* 23 (2020), pp. 3467–3482.

[19] Arnaud Delmotte, Kenichiro Tanaka, Hiroyuki Kubo, Takuya Funatomi, and Yasuhiro Mukaigawa. "Blind watermarking for 3-D printed objects by locally modifying layer thickness". In: *IEEE Transactions on Multimedia* 22.11 (2019), pp. 2780–2791.

[20] Masahiro Suzuki, Pailin Dechrueng, Soravit Techavichian, Piyarat Silapasuphakornwong, Hideyuki Torii, and Kazutake Uehira. "Embedding information into objects fabricated with 3-D printers by forming fine cavities inside them". In: *Electronic Imaging* 29 (2017), pp. 6–9.

[21] Kazutake Uehira, Masahiro Suzuki, Piyarat Silapasuphakornwong, Hideyuki Torii, and Youichi Takashima. "Copyright protection for 3D printing by embedding information inside 3D-printed objects". In: *International workshop on digital watermarking*. Springer. 2016, pp. 370–378.

[22] Zhengxiong Li, Aditya Singh Rathore, Chen Song, Sheng Wei, Yanzhi Wang, and Wenyao Xu. "PrinTracker: Fingerprinting 3D printers using commodity scanners". In: *Proceedings of the 2018 ACM sigsac conference on computer and communications security*. 2018, pp. 1306–1323.

[23] Prusa Research. *Original Prusa 3D printers directly from Josef Prusa*. URL: https://www.prusa3d.com.

[24] Prusa Research. *PrusaSlicer: G-code generator for 3D printers (RepRap, Makerbot, Ultimaker etc.)* 2026. URL: https://github.com/prusa3d/PrusaSlicer.

[25] Makilab A.S.B.L. *Makilab | We are all makers !* URL: https://makilab.org.

[26] Prusa Research. *Imprimante 3D Original Prusa MK4S*. URL: https://www.prusa3d.com/fr/produit/imprimante-3d-original-prusa-mk4s/.

[27] Prusa Research. *Original Prusa MK4S 3D Printer*. URL: https://www.prusa3d.com/product/original-prusa-mk4s-3d-printer-5/#features.

[28] M Ford, M Boucadair, A Durand, P Levis, and P Roberts. *Issues with IP address sharing*. Tech. rep. 2011.

[29] Jeremy Martin, Travis Mayberry, Collin Donahue, Lucas Foppe, Lamont Brown, Chadwick Riggins, Erik C Rye, and Dane Brown. "A study of MAC address randomization in mobile devices and when it fails". In: *arXiv preprint arXiv:1703.02874* (2017).

[30] K Davis, B Peabody, and P Leach. *Rfc 9562: Universally unique identifiers (uuids)*. 2024.

[31] Prusa Research. *Prusa Connect*. URL: https://www.prusa3d.com/fr/p/prusa-connect/.

[32] Tyler Bell, Beiwen Li, and Song Zhang. "Structured light techniques and applications". In: *Wiley encyclopedia of electrical and electronics engineering* (1999), pp. 1–24.

[33] Rubén Usamentiaga, Pablo Venegas, Jon Guerediaga, Laura Vega, Julio Molleda, and Francisco G Bulnes. “Infrared thermography for temperature measurement and non-destructive testing”. In: *Sensors* 14.7 (2014), pp. 12305–12348.

[34] Argonne National Laboratory. *TomoPy Documentation*. URL: https://tomopy.readthedocs.io/en/stable/.

[35] David Caltrider, Abhishek Gupta, and Koushik Tripathy. “Evaluation of visual acuity”. In: *StatPearls [Internet]*. StatPearls Publishing, 2024.

[36] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. Fact 2.27, available at https://cacr.uwaterloo.ca/hac/about/chap2.pdf. CRC Press, 1997. Chap. 2.

[37] William Wesley Peterson and Daniel T Brown. “Cyclic codes for error detection”. In: *Proceedings of the IRE* 49.1 (1961), pp. 228–235.

[38] Philip Koopman and Tridib Chakravarty. “Cyclic redundancy code (CRC) polynomial selection for embedded networks”. In: *International Conference on Dependable Systems and Networks, 2004*. IEEE. 2004, pp. 145–154.

[39] Juan R Hernandez, Jean-Francois Delaigle, and Benoît MM Macq. “Improving data hiding by using convolutional codes and soft-decision decoding”. In: *Security and Watermarking of Multimedia Contents II*. Vol. 3971. SPIE. 2000, pp. 24–47.

List of Figures

List of Tables

Appendix A

Code

Validation Pipeline

gcode_to_voxels.py

```
def parse_gcode(path, pitch=0.2, nozzle_diameter=0.4, layer_height=0.2):

    x, y, z, e = 0.0, 0.0, 0.0, 0.0
    relative_e = False
    moves = []

    # Bounding box
    x_min, x_max = float('inf'), float('-inf')
    y_min, y_max = float('inf'), float('-inf')
    z_min, z_max = float('inf'), float('-inf')

    with open(path, 'r') as f:

        printing_started = False

        for line in f:

            raw_line = line.strip()

            # Detects the start of printing after the purge line
            if ';LAYER_CHANGE' in raw_line:
                printing_started = True

            if raw_line.startswith('M83'):
                relative_e = True
                continue

            if raw_line.startswith('M82'):
                relative_e = False
                continue

            if raw_line.startswith('G92') and 'E' in raw_line:
                e = 0.0
                continue

            if not printing_started:
                continue

            line = line.split(';')[0].strip()
            if not line:
                continue

```

```
            # Handles extrusion mode absolue/relative
            if line == 'M83':
                relative_e = True
                continue

            if line == 'M82':
                relative_e = False
                continue

            # Resets the extruder
            if line.startswith('G92'):
                if 'E' in line:
                    e = 0.0
                continue

            if not (line.startswith('G1') or line.startswith('G0')):
                continue

            # Parses the X, Y, Z and E parameters
            params = {}
            for token in line.split()[1:]:
                if token and token[0] in 'XYZE':
                    try:
                        params[token[0]] = float(token[1:])
                    except ValueError:
                        pass

            new_x = params.get('X', x)
            new_y = params.get('Y', y)
            new_z = params.get('Z', z)
            new_e_raw = params.get('E', e if not relative_e else 0.0)
            new_e = (e + new_e_raw) if relative_e else new_e_raw

            # Extrusion segments
            if new_e > e and ('X' in params or 'Y' in params):
                moves.append((x, y, new_x, new_y, z))
                x_min = min(x_min, x, new_x)
                x_max = max(x_max, x, new_x)
                y_min = min(y_min, y, new_y)
                y_max = max(y_max, y, new_y)
                z_min = min(z_min, z)
                z_max = max(z_max, z)

            x, y, z, e = new_x, new_y, new_z, new_e

    # Creates the voxel grid
    margin = 2
    nx = int(round((x_max - x_min) / pitch)) + 2 * margin + 1
    ny = int(round((y_max - y_min) / pitch)) + 2 * margin + 1
    nz = int(round((z_max - z_min) / layer_height)) + 2 * margin + 1
    print(f"Voxels grid (Z, Y, X) : ({nz}, {ny}, {nx})")

    vol = np.zeros((nz, ny, nx), dtype=np.float32)

    # Converts the radius into voxels
    r = max(1, int(round((nozzle_diameter / 2) / pitch)))

    # Rasterises the segments
    for x0, y0, x1, y1, z_level in moves:
        seg_len = np.sqrt((x1 - x0) ** 2 + (y1 - y0) ** 2)
        n_steps = max(1, int(seg_len / (pitch * 0.5)))

        iz = int(round((z_level - z_min) / layer_height)) + margin
```

```
106
107         for i in range(n_steps + 1):
108             t = i / n_steps
109             px = x0 + t * (x1 - x0)
110             py = y0 + t * (y1 - y0)
111
112             ix = int(round((px - x_min) / pitch)) + margin
113             iy = int(round((py - y_min) / pitch)) + margin
114
115             for dy in range(-r, r + 1):
116                 for dx in range(-r, r + 1):
117                     vx, vy = ix + dx, iy + dy
118                     if 0 <= vx < nx and 0 <= vy < ny and 0 <= iz < nz:
119                         vol[iz, vy, vx] = 1.0
120
121     return vol, margin
```

Listing A.1: G-code parsing: tracking nozzle position and extrusion state (excerpt of `parse_gcode()` in `gcode_to_voxels.py`)

`test_tomopy.py`

```
1  if __name__ == '__main__':
2
3      # 1. Load the test volume
4      vol = tomopy.misc.phantom.shepp3d(size=64)
5      print("Volume shape:", vol.shape)
6
7      # 2. Define the projection angles (from 0 to 180 degrees)
8      angles = tomopy.angles(180)
9
10     # 3. Simulate the X-ray projections
11     proj = tomopy.project(vol, angles)
12     print("Projections shape:", proj.shape)
13
14     # 4. Reconstruct the volume from the projections
15     recon = tomopy.recon(proj, angles, algorithm='gridrec')
16     recon = tomopy.circ_mask(recon, axis=0, ratio=0.95)
17     print("Reconstruction shape:", recon.shape)
```

Listing A.2: Sanity-checking the TomoPy pipeline on its own bundled test phantom (Figure 4.5)

Method 1: Whole-Layer Orientation Modulation

Encoding

```
1  const PrintRegionConfig &region_config = m_regions[surface_fill.region_id]->
       region().config();
2  long bottom_solid_layers = std::max(0, region_config.bottom_solid_layers.value);
3
4  if (!watermark.bits.empty() && (long)f->layer_id >= bottom_solid_layers) {
5      long effective_layer = (long)f->layer_id - bottom_solid_layers;
6
7      // Repeats the watermark as long as infill layers are available
8      if (effective_layer > 0) {
9          long bit_index = (effective_layer - 1) % (long)watermark.bits.size();
10         if (watermark.bits[bit_index] == 1)
11             f->angle += float(Geometry::deg2rad(0.573));
12     }
13 }
```

Listing A.3: Method 1's encoding step (excerpt of `Layer::make_fills()` in `Fill.cpp`)

```cpp
#include <random>

static uint64_t generate_watermark_identifier()
{
    std::random_device rd;
    return (uint64_t(rd()) << 32) | rd();
}
```

Listing A.4: Generation of a random 64-bit opaque identifier

```cpp
struct WatermarkData {

    uint64_t          identifier;
    std::vector<int> bits;

    WatermarkData() {

        identifier = generate_watermark_identifier();
        bits = build_watermark_payload(identifier);

    }
};
```

Listing A.5: The `WatermarkData` structure instantiated in `Layer::make_fills()`

```cpp
static constexpr uint16_t WATERMARK_CRC16_POLY = 0x90D9;

static uint16_t compute_watermark_checksum(uint64_t identifier) {
  uint16_t crc = 0;

  auto process_bit = [&crc](int bit) {
    uint16_t leading = (crc >> 15) & 1u;
    crc = uint16_t((crc << 1) | (bit & 1));
    if (leading)
      crc ^= WATERMARK_CRC16_POLY;
  };

  for (int i = 63; i >= 0; --i)
    process_bit(int((identifier >> i) & 1u));

  // 16 bits of zero padding
  for (int i = 0; i < 16; ++i)
    process_bit(0);

  return crc;
}

std::vector<int> build_watermark_payload(uint64_t identifier) {
  std::vector<int> bits;
  bits.reserve(80);

  for (int i = 63; i >= 0; --i)
    bits.push_back(int((identifier >> i) & 1u));

  uint16_t checksum = compute_watermark_checksum(identifier);
  for (int i = 15; i >= 0; --i)
    bits.push_back(int((checksum >> i) & 1u));

  return bits;
}
```

Listing A.6: Computation of the CRC-16 checksum and assembly of the 80-bit watermark payload

Decoding

```
# Detecting the differences between layers
mean_matrix = []
for i in range(7, len(recon)-8):
    matrix = recon[i-1] - recon[i]
    mean_matrix.append(abs(np.float32(np.mean(matrix))))

values = [float(x) for x in mean_matrix]
indices = range(len(values))

# Recovering the bits from the differences between layers
watermark = []
current_bit = 0
other_bit = 1
for x in values:
    if x < 10 ** (-6):
        watermark.append(current_bit)
    else:
        current_bit, other_bit = other_bit, current_bit
        watermark.append(current_bit)

# Splitting the decoded bit stream into 80-bit copies
size_payload = 80
n_copies = len(watermark) // size_payload

if n_copies == 0:
    raise RuntimeError(
        f"Not enough layers."
    )

copies = [watermark[c * size_payload: (c + 1) * size_payload] for c in range(
    n_copies)]

# Checking the checksum of each copy
valid_copies = [copy for copy in copies if verify_watermark_checksum(copy)]

if len(valid_copies) == 0:
    raise RuntimeError("No valid copy: all of them failed the checksum check.")

copies = valid_copies

# Majority vote
retrieved_payload = []
count = 0
for bit_index in range(size_payload):
    votes = [copy[bit_index] for copy in copies]
    ones = sum(votes)
    zeros = len(votes) - ones
    if ones == zeros:
        count += 1
    retrieved_payload.append(1 if ones > zeros else 0)

# Test payload used to verify that encoding and decoding work correctly
actual_payload = [1,1,0,1,1,0,1,0,...]

error = 0
for i in range(len(actual_payload)):
    if actual_payload[i] != retrieved_payload[i]:
        error += 1
if error == 0:
    print('WATERMARK FOUND')
else:
```

```
    print(f'ERROR IN FINDING THE WATERMARK ({error} character(s) mismatched)')
```

Listing A.7: Method 1's decoding step (excerpt of `decoding_method1.py`)

```
CRC16 = 0x90D9  # Same polynomial as the one used during the encoding

def verify_watermark_checksum(payload_bits):

    crc = 0

    for bit in payload_bits:
        leading = (crc >> 15) & 1
        crc = ((crc << 1) | (bit & 1)) & 0xFFFF
        if leading:
            crc ^= CRC16

    return crc == 0
```

Listing A.8: Verification of a payload copy's checksum in `decoding_method1.py`

Method 2: Per-Line Grid Modulation

Encoding

```
void FillWatermarkGrid::_fill_surface_single(
    const FillParams                 &params,
    unsigned int                    /* thickness_layers */, // Parameter intentionally unused
    const std::pair<float, Point>    &direction,
    ExPolygon                         expolygon,
    Polylines                        &polylines_out)
{
    // Convert the base line spacing to scaled (integer) coordinates
    coord_t base_spacing = scaled<coord_t>(this->spacing);
    if (base_spacing <= 0)
        return;

    // Density is split in two: the vertical and horizontal families each contribute half
    double  density_per_family = std::max(0.0001, double(params.density) / 2.0);
    coord_t e = coord_t(double(base_spacing) / density_per_family);
    if (e <= 0)
        return;

    // Compute the bounding box
    expolygon.rotate(-direction.first);
    BoundingBox bbox = expolygon.contour.bounding_box();
    if (!bbox.defined)
        return;

    // Width/height of the area to fill
    coord_t Wx = bbox.max.x() - bbox.min.x();
    coord_t Wy = bbox.max.y() - bbox.min.y();

    // Number of lines per family
    long    N_x = std::max(1L, long(std::floor(double(Wx) / double(e))) + 1);
    long    N_y = std::max(1L, long(std::floor(double(Wy) / double(e))) + 1);

    // Margin so the lines are centered in the area rather than flush with one edge
    coord_t margin_x = coord_t((double(Wx) - double(N_x - 1) * double(e)) / 2.0);
```

```
    coord_t margin_y = coord_t((double(Wy) - double(N_y - 1) * double(e)) / 2.0)
    ;

    // Number of solid bottom layers to skip before sparse infill (and the
    watermark) begins
    long bottom_solid_layers = print_region_config ? std::max(0,
    print_region_config->bottom_solid_layers.value) : 0;
    bool layer_valid       = this->layer_id != size_t(-1) && long(this->layer_id
    ) >= bottom_solid_layers;
    long effective_layer   = layer_valid ? long(this->layer_id) -
    bottom_solid_layers : 0;

    // Convert the tilt angle from degrees to radians
    float tilt_rad = float(double(watermark_angle_deg) * M_PI / 180.0);

    // Build both line families via build_family() (once per family)
    Polylines polylines_src;
    build_family(true,  bbox, e, N_x, N_y, margin_x, layer_valid,
    effective_layer, tilt_rad, polylines_src); // vertical lines
    build_family(false, bbox, e, N_x, N_y, margin_y, layer_valid,
    effective_layer, tilt_rad, polylines_src); // horizontal lines

    // Append to the output
    Polylines polylines = intersection_pl(polylines_src, offset(expolygon,
    scale_(0.02)));
    for (Polyline &pl : polylines)
        pl.rotate(direction.first);
    append(polylines_out, std::move(polylines));
}
```

Listing A.9: Generating the watermark grid for one surface region (excerpt of `FillWatermarkGrid.cpp`)

```
void FillWatermarkGrid::build_family(
    bool                vertical,
    const BoundingBox &bbox,
    coord_t             e,
    long                N_x,
    long                N_y,
    coord_t             margin_fixed,
    bool                layer_valid,
    long                effective_layer,
    float               tilt_rad,
    Polylines          &polylines_src) const
{
    // Sets up the bounds depending on whether we're building vertical or
    horizontal lines
    coord_t fixed_min   = vertical ? bbox.min.x() : bbox.min.y();
    coord_t running_min = vertical ? bbox.min.y() : bbox.min.x();
    coord_t running_max = vertical ? bbox.max.y() : bbox.max.x();

    // Number of lines in this family
    long N = vertical ? N_x : N_y;

    // Horizontal lines are indexed right after the N_x vertical ones, hence
    this offset
    long family_offset = vertical ? 0 : N_x;

    // Length the lines need to span
    coord_t length = running_max - running_min;
    if (length < 0)
        return;

    // Safety margin
    coord_t margin_tilt = coord_t(std::ceil(std::abs(std::tan(double(tilt_rad)))
```

```
     * double(length) / 2.0)) + SCALED_EPSILON;

    // Loop over every line in this family
    for (long a = 1; a <= N; ++a) {

        // Coordinate (X for vertical lines, Y for horizontal lines) of this line
        coord_t c = fixed_min + margin_fixed + coord_t(a - 1) * e;

        // Decide whether this line encodes a 0 or a 1
        int bit = 0;
        if (layer_valid) {
            if (effective_layer == 0) {
                // First infill layer: forced all-zero calibration reference
                bit = 0;
            } else if (effective_layer == 1) {
                // Second infill layer: forced all-one calibration reference
                bit = 1;
            } else if (!watermark_bits.empty()) {
                // Index of the bit to encode, within the watermark message
                long position = (effective_layer-2) * (N_x + N_y)
                              + family_offset + (a - 1);
                if (position >= 0) {
                    long idx = position % long(watermark_bits.size());
                    bit = watermark_bits[size_t(idx)] ? 1 : 0;
                }
            }
        }

        // Creates the segment's two endpoints
        Point p0, p1;
        if (vertical) {
            p0 = Point(c, running_min - margin_tilt);
            p1 = Point(c, running_max + margin_tilt);
        } else {
            p0 = Point(running_min - margin_tilt, c);
            p1 = Point(running_max + margin_tilt, c);
        }

        // If the bit is 1, rotate the endpoints around the segment's center
        if (bit) {
            Point center((p0.x() + p1.x()) / 2, (p0.y() + p1.y()) / 2);
            p0.rotate(double(tilt_rad), center);
            p1.rotate(double(tilt_rad), center);
        }

        // Builds a Polyline from the two points and appends it to the output vector
        Polyline pl;
        pl.points.reserve(2);
        pl.points.push_back(p0);
        pl.points.push_back(p1);
        polylines_src.emplace_back(std::move(pl));
    }
}
```

Listing A.10: Placing and tilting one family of grid lines (excerpt of `FillWatermarkGrid.cpp`)

Decoding

```
# 3D matrix filtering
for z in range(len(recon)):
    for i in range(len(recon[z])):
```

```
        for j in range(len(recon[z][i])) :
            if recon[z][i][j] < 0.35 :
                recon[z][i][j] = 0

# Creating the reference for the peaks
reference = []
for i in range(len(recon[6])):
    line = []
    for j in range(len(recon[6][i])) :
        diff = abs(recon[6][i][j]-recon[7][i][j])
        line.append(diff)
    reference.append(line)

# Peak reference for the columns:
col_peak_ref = []
for j in range(len(reference[0])) :
    sum = 0
    for i in range(len(reference)) :
        sum += reference[i][j]
    col_peak_ref.append(sum)

# Peak reference for the rows:
line_peak_ref = []
for i in range(len(reference)) :
    sum = 0
    for j in range(len(reference[i])) :
        sum += reference[i][j]
    line_peak_ref.append(sum)

regions_col = detect_peaks_reference(col_peak_ref)
regions_line = detect_peaks_reference(line_peak_ref)

watermark = []

# Iterating over each infill layer
for z in range(8, len(recon)-6) :

    col_peaks = []
    line_peaks = []

    abs_diff_matrix = []
    for i in range(len(recon[z])):
        line = []
        for j in range(len(recon[z][i])) :
            diff = abs(recon[6][i][j]-recon[z][i][j])
            line.append(diff)
        abs_diff_matrix.append(line)

    for j in range(len(abs_diff_matrix[0])) :
        sum = 0
        for i in range(len(abs_diff_matrix)) :
            sum += abs_diff_matrix[i][j]
        col_peaks.append(sum)

    for i in range(len(abs_diff_matrix)) :
        sum = 0
        for j in range(len(abs_diff_matrix[i])) :
            sum += abs_diff_matrix[i][j]
        line_peaks.append(sum)

    x_indexes = compare_peaks(col_peak_ref, col_peaks, regions_col)
    y_indexes = compare_peaks(line_peak_ref, line_peaks, regions_line)

```

```
    for i in range(len(x_indexes)) :
        if x_indexes[i] :
            watermark.append(1)
        else :
            watermark.append(0)

    for i in range(len(y_indexes)) :
        if y_indexes[i] :
            watermark.append(1)
        else :
            watermark.append(0)
```

Listing A.11: 3D matrix filtering and watermark extraction via peak comparison (excerpt of `decoding_method2.py`)

```
def detect_peaks_reference(reference, k=1.5, gap=2):
    """
    Returns a list of tuples (start, end), one per peak detected in the
    reference
    """

    reference = np.asarray(reference, dtype=float)
    med = np.median(reference)
    mad = np.median(np.abs(reference - med))
    threshold = med + k * mad * 1.4826

    above = reference > threshold
    regions = []
    start = None
    for i, v in enumerate(above):
        if v and start is None:
            start = i
        elif not v and start is not None:
            regions.append((start, i))
            start = None
    if start is not None:
        regions.append((start, len(reference)))

    fusion = []
    for r in regions:
        if fusion and r[0] - fusion[-1][1] <= gap:
            fusion[-1] = (fusion[-1][0], r[1])
        else:
            fusion.append(list(r))

    return [tuple(r) for r in fusion]
```

Listing A.12: Peak region detection in the reference signal (excerpt of `decoding_method2.py`)

```
def compare_peaks(reference, target, regions, gap_threshold=0.4):
    """
    Compares the mean of `reference` and `target` over each interval.
    If the relative gap exceeds `threshold_gap`, the peak is considered absent.
    Returns a list of booleans, one per peak, in order.
    """

    reference = np.asarray(reference, dtype=float)
    target = np.asarray(target, dtype=float)

    results = []
    for start, end in regions:
        mean_ref = reference[start:end].mean()
        mean_target = target[start:end].mean()
        relative_gap = abs(mean_ref - mean_target) / mean_ref
```

```
16          results.append(relative_gap <= gap_threshold)
17      return results
```

Listing A.13: Reference/target peak comparison to validate watermark presence (excerpt of `decoding_method2.py`)

Appendix B

Simulation Values

Tilting angle

Displacement	1 cm cube	2 cm cube	3 cm cube
60 μm	0.688°	0.343°	0.229°
70 μm	0.802°	0.401°	0.267°
80 μm	0.917°	0.458°	0.306°
90 μm	1.031°	0.516°	0.344°
100 μm	1.146°	0.573°	0.382°
110 μm	1.260°	0.630°	0.420°
120 μm	1.375°	0.688°	0.458°
130 μm	1.489°	0.745°	0.497°
140 μm	1.604°	0.802°	0.535°

Table B.1: Tilting angle from Equation (3.1)

Method 1

Displacement	1 cm cube	2 cm cube	3 cm cube
60 μm	/	1	1
70 μm	/	1	1
80 μm	/	1	1
90 μm	/	1	1
100 μm	/	1	1
110 μm	/	1	1
120 μm	/	1	1
130 μm	/	1	1
140 μm	/	1	1

Table B.2: Number of complete payload copies isolated by Method 1's decoder

Displacement	1 cm cube	2 cm cube	3 cm cube
60 μm	/	100 %	100 %
70 μm	/	100 %	100 %
80 μm	/	100 %	100 %
90 μm	/	100 %	100 %
100 μm	/	100 %	100 %
110 μm	/	100 %	100 %
120 μm	/	100 %	100 %
130 μm	/	100 %	100 %
140 μm	/	100 %	100 %

Table B.3: Percentage of the detected copy that passed its 16-bit CRC checksum

Displacement	1 cm cube	2 cm cube	3 cm cube
60 μm	/	0.00 %	0.00 %
70 μm	/	0.00 %	0.00 %
80 μm	/	0.00 %	0.00 %
90 μm	/	0.00 %	0.00 %
100 μm	/	0.00 %	0.00 %
110 μm	/	0.00 %	0.00 %
120 μm	/	0.00 %	0.00 %
130 μm	/	0.00 %	0.00 %
140 μm	/	0.00 %	0.00 %

Table B.4: BER of Method 1's decoding scheme (after checksum)

Method 2

Displacement	1 cm cube	2 cm cube	3 cm cube
60 μm	2	7	19
70 μm	2	6	19
80 μm	1	8	22
90 μm	1	8	20
100 μm	1	8	19
110 μm	1	8	22
120 μm	1	8	20
130 μm	1	8	20
140 μm	1	8	20

Table B.5: Number of complete payload copies isolated by Method 2's decoder

Displacement	1 cm cube	2 cm cube	3 cm cube
60 μm	33 bits	/	/
70 μm	35 bits	0 bit	/
80 μm	/	0 bit	6 bits
90 μm	/	0 bit	0 bit
100 μm	/	0 bit	/
110 μm	/	0 bit	8 bits
120 μm	/	0 bit	0 bit
130 μm	/	0 bit	0 bit
140 μm	/	0 bit	0 bit

Table B.6: Number of tied bits resolved by the majority vote of Method 2's decoding scheme (only applicable when the copy count is even)

Displacement	1 cm cube	2 cm cube	3 cm cube
60 μm	0 %	0 %	0 %
70 μm	0 %	0 %	0 %
80 μm	0 %	0 %	0 %
90 μm	0 %	100 %	0 %
100 μm	0 %	100 %	0 %
110 μm	0 %	100 %	0 %
120 μm	0 %	100 %	70 %
130 μm	0 %	100 %	100 %
140 μm	0 %	100 %	100 %

Table B.7: Percentage of the detected copy that passed its 16-bit CRC checksum

Displacement	1 cm cube	2 cm cube	3 cm cube
60 μm	53.75 %	50.00 %	41.25 %
70 μm	52.5 %	50.00 %	43.75 %
80 μm	6.25 %	11.25 %	50.00 %
90 μm	6.25 %	0.00 %	27.50 %
100 μm	10.00 %	0.00 %	53.75 %
110 μm	11.25 %	0.00 %	53.75 %
120 μm	11.25 %	0.00 %	0.00 %
130 μm	11.25 %	0.00 %	0.00 %
140 μm	10.00 %	0.00 %	0.00 %

Table B.8: BER of Method 2's decoding scheme (after checksum)

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**