frontiers | Frontiers in Computer Science

TYPE Original Research
PUBLISHED 11 November 2025
DOI 10.3389/fcomp.2025.1643075

OPEN ACCESS

EDITED BY
Vincenzo Arceri,
University of Parma, Italy

REVIEWED BY
Tarun Kumar Vashishth,
IIMT University, India
Lucila Bento,
Rio de Janeiro State University, Brazil

*CORRESPONDENCE
Maikel Lázaro Pérez Gort
maikel.perezgort@unive.it

RECEIVED 07 June 2025
ACCEPTED 27 October 2025
PUBLISHED 11 November 2025

CITATION
Pérez Gort ML (2025) Input parameters authentication through dynamic software watermarking. *Front. Comput. Sci.* 7:1643075. doi: 10.3389/fcomp.2025.1643075

Input parameters authentication through dynamic software watermarking

Maikel Lázaro Pérez Gort*

Department of Environmental Sciences, Informatics and Statistics, Ca' Foscari University of Venice, Venice, Italy

Modern civilization relies on computers and the Internet. Web services and microservices make many processes more accessible, often without users realizing the extent of their dependency. As digitalization spreads, the integrity of input parameters used by programmed methods becomes crucial for generating accurate and reliable outcomes, which are essential for the proper functioning of society. This paper introduces a dynamic software watermarking approach designed to validate the authenticity of input parameters in high-level programming language functions. The proposed approach operates without interfering with software functionalities and is resilient to code optimization, obfuscation, and other transformations. The experimental results demonstrate the robustness of our method, ensuring 100% accuracy in detecting tampering with parameter values across all test cases.

KEYWORDS

software watermarking, parameter authentication, software verification, dynamic watermarking, security

1 Introduction

Since the creation of the Internet, the use of web services has steadily increased. Today, they enable businesses to expand their reach and improve profitability by offering platform access to customers worldwide. Our reliance on web services has become so ingrained that users often utilize them without even realizing it. However, this dependence has a downside: the widespread tendency to accept information from online sources without questioning its authenticity.

As more services and digital assets are deployed in the cloud, the risks of tampering increases. Data can be altered and communications intercepted, allowing intruders to manipulate content or modify parameters passed to web services, which can result in unauthorized, inaccurate, or misleading information.

Such malicious actions can compromise the authenticity of web service outcomes, degrade service quality, and damage an organization's reputation and competitiveness. The severity of these incidents depends on the organization's role and responsibilities. For example, consequences can range from providing incorrect navigation directions to overlooking critical medical conditions in hospitals, which can potentially lead to life-threatening consequences. Therefore, ensuring the correct and reliable operation of web services is always paramount.

A concrete incident that illustrates the risks described above is the Panera Bread case (April 2018), in which millions of customer records were exposed via an API endpoint.[1]

1 *"Panerabread.com Leaks Millions of Customer Records"*, KrebsOnSecurity, available at https://krebsonsecurity.com/2018/04/panerabread-com-leaks-millions-of-customer-records/.

Another example is the Optus data breach (September 2022), which underscored that an unauthenticated, Internet-facing API, combined with insufficient parameter handling and authorization, can permit large-scale data access[2, 3]. Beyond individual breaches, studies of HTTP Parameter Pollution (HPP) show how duplicate or injected parameters can alter back-end behavior. A large-scale measurement revealed HPP issues affecting high-profile sites, including Google, PayPal, and Microsoft (Balduzzi et al., 2011). E-commerce provides further evidence: publicly disclosed bug bounty reports and practitioner surveys document price manipulation via parameter tampering in checkout flows (e.g., altering amount/discount fields), highlighting the persistence of business-logic vulnerabilities despite conventional input filtering and web application firewalls (WAFs).[4]

An important number of web services are implemented using high-level programming languages. The primary objective of this work is to propose an approach for validating the authenticity of input parameters in high-level programming language functions that are crucial to web service development, thereby protecting business processes even when other security measures fail, as prior cases illustrate. The study performed in this research evaluates the strengths and limitations of existing solutions and explores the potential benefits of integrating them. Particular attention is given to dynamic software watermarking, emphasizing its adaptability to software runtime behavior, with a focus on the role of managed data.

This work is guided by three research questions: (i) - Does the proposed approach detect tampering of input parameters with a high true-positive rate and low false-positive rate across representative workloads? (ii) - Does integrating the approach preserve the functional output of protected functions when the input parameters are authentic? (iii) - Does integrating the approach impose a bounded, predictable overhead in code size and execution time, with runtime scaling linearly in input size across representative workloads? (iv) - Are authentication decisions robust to compiler/interpreter optimizations, code obfuscation, common refactoring, and heterogeneous runtime environments?

With the proposed approach, execution is blocked when parameters are flagged as non-authentic, thereby preventing computation on tainted values, even if the function would otherwise run correctly. The approach is validated through a Python implementation; however, the solution can be extended to various environments where processes are implemented using high-level programming languages. In doing so, the integrity of web services and microservices can be safeguarded.

2 *"Optus to answer privacy court case stemming from 2022 data breach"*, by Ry Crozier, available at https://www.itnews.com.au/news/optus-to-answer-privacy-court-case-stemming-from-2022-data-breach-619417.

3 *"Australian Information Commissioner takes civil penalty action against Optus"*, Office of the Australian Information Commissioner (OAIC), available at https://www.oaic.gov.au/news/media-centre/australian-information-commissioner-takes-civil-penalty-action-against-optus.

4 *"Hunting the 6 most common price manipulation vulnerabilities in e-commerce"*, Intigriti, available at https://www.intigriti.com/blog/news/top-6-price-manipulation-vulnerabilities-ecommerce.

The rest of this paper is organized as follows: Section 2 introduces the preliminaries of software watermarking. Section 3 presents related work as part of potential solutions to address the stated problem and outlines the leading dynamic software watermarking techniques proposed so far. Section 4 provides details of the proposed approach, and Section 5 presents the results of the experiments conducted to validate it. Section 6 outlines the added value of our approach, which extends beyond conventional integrity mechanisms. Finally, Section 7 concludes.

2 Preliminaries

Digital watermarking techniques enable the verification of ownership, detection of data tampering, validation of content authenticity, tracing of data copies, and enforcement of licensing conditions (Vivekananthan et al., 2021). They work by embedding a watermark into the protected digital asset, also known as the carrier. The watermark consists of bits, each constituting a mark, and can store copyright information or be generated from the content to verify its authenticity.

A watermarking technique consists of two basic processes: watermark embedding and extraction. Extraction may occur when there is suspicion of copyright violation, false ownership claims, data tampering, or whenever required to trace users violating purchase contracts, also known as traitors. Watermarking techniques have proven highly effective in protecting various types of digital assets, including multimedia data (such as images, audio, or video), relational data, graphs, documents, and software.

2.1 Software watermarking architecture

The architecture defining software watermarking approaches is characterized by a watermark W embedded into a program P in such a way that W can be reliably located and extracted from P, even after P has undergone semantics-preserving transformations such as code optimization and obfuscation. The watermark W should be stealthy, have a high data rate, and not affect the performance of P. Additionally, it must be demonstrated that the presence of W in P is a result of deliberate actions (Collberg and Thomborson, 1998; Thomborson et al., 2004).

The software watermarking architecture is driven by the implementation of the embedding and extraction functions, denoted as $\mathcal{I}(\cdot)$ and $\mathcal{E}(\cdot)$, respectively.[5] The embedding is featured by $P' = \mathcal{I}(W, P, k)$, where $\mathcal{I}(\cdot)$ inputs are the watermark W, the program P, and the program's owner cryptography secret key k. As a result, the watermarked version of the program, denoted as P', is obtained. The extraction function, on the other hand, is featured by $W' = \mathcal{E}(P', k)$, where W' is the watermark extracted from P'. For the final assertion, ownership or lack of tampering is proven if both watermarks are similar enough, according to $W \equiv W'$. The operator $\equiv$ considers inconsistencies in detection due to benign operations or attacks that do not impact software performance or the watermark recognition.

5 Some techniques define the watermark encoder and decoder, to build W before its embedding and after its extraction, respectively (Bento et al., 2019).

TABLE 1 Main differences between dynamic and static software watermarking.

Criterion	Static watermarking	Dynamic watermarking
Embedding	Watermark embedded into static components (code or binaries) at design/compile time.	Watermark embedded during runtime, based on the application's state or inputs.
Tied to execution	No, the watermark exists independently of runtime.	Yes, they rely on the software running to produce or verify the watermark.

2.2 Static and dynamic approaches

Software watermarking techniques can be classified into two categories: static and dynamic. Static techniques embed the watermark directly into the software's static artifacts, such as its source code, compiled binary, data files, or the application executable itself. There, the watermark remains as part of the software regardless of whether it is running or not. On the other hand, dynamic watermarking techniques store the watermark in the program execution state (Wu et al., 2024). Dynamic watermarks can be processed, verified, or executed during the runtime of the software or application. They are tied to the behavior or state of the application as it operates (Pieprzyk, 1999; Dey et al., 2019). Table 1 resumes the differences between dynamic and static software watermarking.

According to Collberg and Thomborson (1998), there are three types of dynamic software watermarking approaches: (i) Easter egg watermarking, (ii) data structure watermarking, and (iii) execution trace watermarking. Easter egg watermarking is based on activating a code fragment when the application is run using an unusual input. As a result, the software performs an action immediately perceptible to its users, such as displaying a copyright message, which makes the watermark's presence evident. On the other hand, data structure approaches are based on embedding the watermark in a specific program state, which is obtained when a particular input sequence is used. In this approach, the watermark is extracted by examining the current values of the variables after the input sequence has ended. The extraction can be done using a dedicated watermark extraction routine linked to the executing program or by running the program under a debugger. Given that no output is ever produced, it is not evident to an adversary when the special input sequence has been entered. Finally, execution trace approaches embed the watermark into instructions and/or addresses, as the program is run with a particular input. In that case, the watermark is extracted by monitoring (statistical) properties of the address trace and/or the sequence of operators executed.

This research aligns with dynamic software watermarking, particularly dynamic data structure watermarking, as it proposes an approach that generates and validates watermarks as part of the software's runtime behavior to verify the authenticity of input parameters for functions. Since static watermarking requires embedding the watermark in the source code or compiled software itself, which does not contribute to solving the targeted problem, it is excluded from this work.

2.3 Watermarking requirements

When designing a software watermarking approach, certain conditions must be fulfilled, regardless of whether the technique is static or dynamic. As previously mentioned, the watermark W must be embedded into the program P, ensuring its detectability despite benign operations such as code optimization and obfuscation. Besides detectability, other requirements must be fulfilled, such as stealthiness and a high data rate (Ma et al., 2019). These are all requirements that work in favor of the watermarking tradeoff between data rate (or capacity), robustness, and stealthiness (or imperceptibility) (Cox et al., 2007; Pérez Gort et al., 2021). It is paramount to avoid collusion with the program performance by ensuring that the watermark does not alter its size or significantly impact its execution time (Alitavoli et al., 2013; Kumar et al., 2015).

In addition to the above requirements, this research also considers the blindness property, which applies broadly to watermarking irrespective of the protected digital asset. A watermarking technique is blind if the watermark extraction does not require access to either the original (unwatermarked) content or the watermark payload itself (Halder et al., 2010; Pérez Gort et al., 2017). Techniques lacking this property are non-blind. Ensuring blindness reduces the risk of unauthorized detection or removal, because the detector does not need to store or transmit the original content or embedding materials, thereby lowering the likelihood of sensitive information leakage.

2.4 Adversary model

Despite the requirements, watermarking techniques must guarantee resilience against attacks aiming at removing the watermark to allow adversaries to claim software ownership or perform tampering without being noticed. According to the related literature, some of the more relevant attacks are Myles and Collberg (2006) and Dalla Preda and Ianni (2024):

- **Subtractive attacks:** Based on subtracting the watermark from the protected software without compromising its performance.
- **Distortive attacks:** Based on distorting the software to compromise the watermark detection before affecting the software functionality.
- **Additive attacks:** Based on inserting a secondary watermark owned by the adversary, expecting to totally or partially overwrite the original watermark, compromising its recognition or making it impossible to prove its temporal precedence.

Other attacks have been defined as particular implementations or extensions of the previously mentioned ones (Dey et al., 2020). For example, *local modification attacks* modify the code similarly to how watermark insertion is performed. Thus, it can be defined as a refined version of a *distortive attack*. The *code reordering attack* modifies the code by reordering large independent pieces of it. The *code addition attack* works by adding a piece of code to the program that should either not be executed or do nothing,

thereby preserving the program's functionality. Additionally, the *code decompilation attack* involves decompiling and recompiling the code. In contrast, the *code compression attack* modifies the code using compression tools, reducing the code size and adding a decompression routine to it (Stern et al., 2000).

Other attacks are more focused on targeting the engines used for watermark identification. This is the case of the *recognition attack*, focused on modifying or disabling the watermark detector to obtain misleading results (Chroni and Nikolopoulos, 2011). There is also the *protocol attack*, which describes cases where the owner lacks complete security over the choice of the recognition (or extraction) engine and the secret keys involved in the watermark synchronization.[6] In such cases, the adversary can proceed by using a fake secret key that purports to prove the existence of his watermark in the unmodified program using the program owner's recognition function (Collberg et al., 2007).

Some attacks focus on the type of watermarks. For example, fingerprinting approaches, which mark different copies of the same asset with distinct watermarks, must be resilient to *collusion attacks*. They are based on combining non-intersecting parts of different copies of the same digital asset, to make it impossible to recognize any of the watermarks embedded in each copy (Dalla Preda and Pasqua, 2019).

Increased resilience to attacks can be achieved by increasing watermark capacity. However, this can compromise its imperceptibility and provoke he technique to interfere with the software's correct functioning. No single technique can guarantee robustness to all attacks. Instead, better results can be obtained by combining different security approaches simultaneously (Bender et al., 1996; Collberg and Thomborson, 1998). Regarding watermarking, the best possible results can be achieved by creating a technique that can withstand the attacks expected in the scenario in which it will be used.

3 Related work

The first software watermarking technique was a static approach based on an algorithm that encoded the watermark by reordering the program's basic blocks (Davidson and Myhrvold, 1996). Since then, several static and dynamic techniques have been proposed, increasing both the complexity and diversity in the field (Hamilton and Danicic, 2010). The first dynamic approach was proposed by Collberg and Thomborson (1998), and was later followed by other methods based on similar principles. Examples include the proposal by Palsberg et al. (2000), which embeds the watermark into dynamic data structures, and the technique by Pieprzyk (1999), which focuses on copyright protection for classified software identities using fingerprinting. Another proposal by Palsberg et al. (2000) aims to protect Java programs with minimal changes in code size and execution time while offering substantial robustness against subtractive, distortive, and additive attacks. Collberg et al. (2004b) introduced a graph-based dynamic watermarking architecture called UWStego. In their work, the authors also presented several metrics to evaluate the effectiveness of various watermarking techniques.

Curran et al. (2004) addressed the practical implications of graph-based dynamic watermarking, with a primary focus on stealth. Tamada et al. (2004) proposed a dynamic approach for detecting the theft of Windows applications, showcasing strong resilience to various attacks. Collberg et al. (2004a) introduced a dynamic path-based method that embeds the watermark in the program's runtime branch structure. Thomborson et al. (2004) developed a tamper-proofing technique called constant encoding, which embeds the watermark into dynamic data structures. Chiru (2005) proposed a dynamic method that watermarks programs by slightly altering their numeric outputs, focusing on intellectual property protection and ownership proof. Myles and Jin (2005) combined code obfuscation with tamper detection to trace the source of illegal redistribution, also enabling pre-packaged, fingerprinted software distribution tied to individual users. Madou et al. (2005) analyzed potential attacks on path-based watermarking, using the proposal by Collberg et al. (2004a) as a case study. Myles and Collberg (2006) explored watermarking Java programs through the use of opaque predicates and reported on both static and dynamic techniques implemented in the Sandmark framework (Collberg et al., 2003).

The variety of techniques highlights the diversity of the field. For instance, Dalla Preda et al. (2008) proposed a method that hides the watermark using semantic instances within loop structures. Techniques by Chan et al. (2012) and Tian et al. (2015) leverage software birthmarks–unique program characteristics–for detecting software plagiarism. Other approaches, such as that by Ma et al. (2015), utilize return-oriented programming to embed watermarks in the data region, thereby avoiding code-analysis-based attacks. Chen et al. (2017, 2018) focused on strengthening the connection between the watermark and the program's semantics, addressing a common weakness in many techniques.

Some proposals are defined as chaos-based approaches. For example, Ke-xin et al. (2009) used chaotic encryption to split the watermark using the Shamir threshold scheme (Desmedt, 2025), which allows watermark recovery from partial information and enhances resilience. citechionis2013dynamic combined opaque predicates with graph theory to increase complexity. Alrehily and Thayananthan (2018) proposed a return-oriented programming technique based on gadget analysis, using their categories and quantity as central research elements. Wang et al. (2018) introduced an approach that exploits exception handling–a common but difficult-to-remove component of software–to resist additive and subtractive attacks.

Despite the wide variety, most dynamic techniques fall into two main categories: (i) path-based and (ii) graph-based approaches. Notable examples of path-based techniques include those by Gupta and Pieprzyk (2006, 2007). Graph-based techniques are more numerous, including proposals by Patel and Pattewar (2014), Chionis et al. (2014), and Chen et al. (2016).

Today, the boundaries of software watermarking have significantly expanded, with new goals and applications. For instance, Lee et al. (2023) proposed a technique to identify code generated by machine learning models, and Kim et al. (2023) developed a method for ownership protection of smart contracts. Nevertheless, despite the range of problems addressed by dynamic

6 Watermark synchronization refers to both the embedding and extraction processes together (Pérez Gort et al., 2020).

watermarking, no technique has yet been proposed to verify the authenticity of input parameters.

4 Proposed approach

The technique proposed in this work does not rely on modifying the source code or binary files of the software to embed the watermark. Instead, it leverages the dynamic structures created by the software when it is executed to synchronize the watermark and verify the authenticity of the parameters received as input by the program's functions. Details about the approach are presented in this section.

4.1 Architecture of data structure approaches

Dynamic data structure watermarking requires setting up the program in a way that allows specific values representing the watermark to be computed at runtime. These computations are triggered only under specific execution conditions, such as certain inputs or execution paths. The watermark is encoded into the control or data flow, which often requires the insertion of new functions or procedures, additional data structures, or conditional logic to hide and trigger watermark behavior. Although the watermark is revealed dynamically during execution, setting up that watermark requires injecting code. This can be improved if the injected code is done in a stealthy or obfuscated manner. Generally, the modifications made to the program code are non-trivial, as they are designed to be challenging to detect and remove. Considering this, a general description of the architecture of data structure dynamic software watermarking techniques is depicted in Figure 1.

In Figure 1, P represents the program acting as a carrier. Inside P, the block named "Watermark Encoding" represents the scattered code injected and obfuscated to build W after a particular set of inputs I is given to the program. Notice that in this case, we model the secret key k as part of I, which is not necessarily always the case.

After P is executed, the *Controller*, considered an external application, accesses the memory and tries to detect W from it. Finally, a secret code computed using the detected watermark is compared with the one expected to be generated for the correct input, and a final assessment is provided regarding the integrity and authenticity of I.

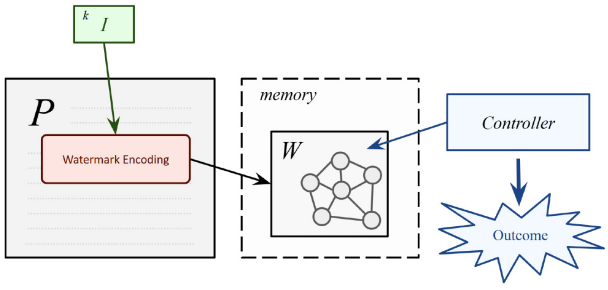

FIGURE 1
General architecture of data structure dynamic software watermarking techniques.

4.2 Architecture of the parameters authentication approach

The architecture of the approach proposed in this work is built based on dynamic data structure watermarking. In this case, the *Controller* is embedded in the program P as scattered and obfuscated code. Since the main goal is to validate I for the function F, P can be stored and deployed in a more secure environment. Nevertheless, this is not a requirement.

The logic of the *Controller* is placed between the "Watermark Encoding" logic and F execution. After an outcome is obtained, if the secret code built with the detected watermark matches the expected value (i.e., the parameters are considered authentic), the process flow is given to F with the input set I. This call is denoted F^I. On the contrary, if the parameters are considered non-authentic, alternative processing is performed, either by ignoring F call or by passing other parameters to it to indicate the required alternative processing. This stage for F calling is defined as F^ϕ. The entire architecture of this approach is depicted in Figure 2.

4.3 Watermark construction

In this work, the watermark is built based on the principles introduced in Collberg and Thomborson (1998). Based on the premise that W is a set of mathematical structures, and p a predicate such as $\forall w \in W : p(w)$, we need to choose p and W such that the probability of $p(x)$ for a random $x \notin W$ is small. Thus, p is typical for W, but atypical outside of it. An example to illustrate the selection of W and p is considering W to be the set of even numbers from 1 to 100 (formally, $W = \{x \in \mathbb{Z}^+ | 1 \leq x \leq 100 \wedge x \bmod 2 = 0\}$), and $p(x)$ the predicate *"x is divisible by 2"*. Considering that, after picking a random value $x \in \{1, ..., 100\}$, if x is odd, $p(x)$ is false. Thus, p is rare outside W.

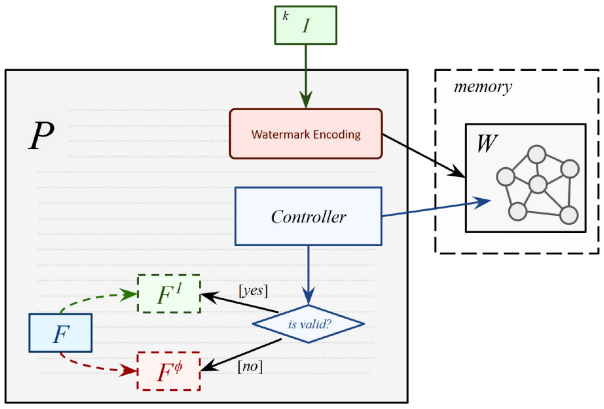

FIGURE 2
Architecture of the input parameters authentication approach.

Although W can be embedded in the topology of any data structure built dynamically in the memory, graphs are particularly a good choice, considering that code that manipulates dynamic graph structures is hard to analyze due to pointer aliasing effects. Due to this, semantics-preserving transformations that make fundamental changes to a graph are more complex to construct (Collberg and Thomborson, 1999).

One of the advantages of targeting high-level programming languages is that there is high coverage to pass values in I that will contribute to the creation of W. Every object can be dynamically created and stored on the heap, and references can be used to develop pointer-like structures. The defined references can be traversed similarly in a graph or another structure storing a different representation of a predefined graph, such as a linked list. This work applies *steganography* principles to hide in I the values that will trigger the dynamic construction of W. Formally, I is composed of a set of input parameters $i_l : l \in [0, |I|)$, along with the secret key k.

4.4 Development of the parameter authentication approach

The construction of the proposed approach requires adjusting to the highest possible level of abstraction, resorting to using references and objects instead of raw pointers and memory, utilizing custom classes to simulate nodes and graphs, among other strategies. All watermarking functionalities embedded into the program must be integrated and dispersed throughout the main logic of the code. Further robustness can be added by obfuscating the code. Nevertheless, higher resilience must be focused on the data structure in memory that stores the watermark.

The parameter authentication approach uses the watermark to verify the authenticity of input parameters, essentially treating the watermark as a covert integrity check. It utilizes a hidden heap-based watermark graph that is only constructed when the program receives the correct input parameters. Later, the program can: (i) decode the watermark at runtime, (ii) verify that the decoded value matches the expected ones, and (iii) proceed only if the input is considered authentic due to the triggering of the correct watermark.

An important feature of the watermark encoding is the addition of fake values into I. They are called *"fake"*, considering that they do not originally belong to I but are introduced to support the construction of the watermark. If these values are not contained in the set of parameters, the instance of I is reported as non-authentic, considering that the correct watermark cannot be reconstructed. The fake values constitute watermark hints, which are initially computed from the binary notation of a secret integer code ζ (denoted by ζ_2) that represents the watermark in numeric format. Next, ζ_2 is used to build Π, which is scattered into I using the secret key k to guarantee the synchronization whenever extraction is required. Finally, when the program receives the parameters, the fake values are extracted from I to rebuild Π and get the code for checking the authenticity of I by building the watermark and comparing it with the original one.

There are different alternatives to synchronize the watermark hints Π in I. The most appropriate choice depends on the data types present in I and the design priorities of the system architecture. Priorities can focus on making hints unnoticeable or ensuring their preservation despite updates to parameter values, which in fact transitions from *steganography* to watermarking principles (Katz et al., 1996; Barán et al., 2001). Nevertheless, our proposal focuses on making the presence of hints unnoticeable, since any change made to the input parameters must compromise the hints so that tampering can be detected. Some of the alternatives to implement the hints synchronization are: (i) positional (index-based) encoding (Katzenbeisser and Petitcolas, 2016), (ii) value modification based on less significant bit (*lsb*) encoding (Cox et al., 2008), (iii) spread spectrum embedding (Barni et al., 2001), (iv) statistical property encoding (Sion et al., 2003), and (v) redundant data slot (Provos and Honeyman, 2003).

We use the case of a function receiving an array of integers to model and validate our approach. We implement the hints synchronization using *lsb* encoding based on difference expansion (Gupta and Pieprzyk, 2009). This approach guarantees high capability for the hints encoding as well as reversibility, which ensures that once the hints are extracted, the parameters are restored to their original values. Consequently, if the parameters are considered authentic, the function can proceed to compute an authentic result that is not affected by the presence of the watermarking mechanism.

A more detailed description of the approach is provided below, outlining the procedures that define the watermark encoding and the controller. Algorithm 1 defines the watermark encoding, using as input the secret key k, the input set I, and the length of the secret code's binary notation ζ_2 (denoted by η). First, it uses k to compute a `seed` utilizing a pseudo-random number generator implemented by the function PRNG$(\cdot)$ (see line 1). Then, the `seed` is used by the function KEYED_PERMUTATION$(\cdot)$ to produce a permutation of the $\lfloor |I|/2 \rfloor$ disjoint pairs in I, which determines the pair order for difference expansion (see line 2). The function returns the keyed deterministic permutation index list $\mathbb{P} = \{p_i \in \mathbb{Z}^+ | 0 \leq i < \lfloor |I|/2 \rfloor\}$. The permutation is reproducible for the same inputs to KEYED_PERMUTATION$(\cdot)$, unpredictable without the seed, and aligns with deterministic pseudorandom function (PRF)-driven derivations in cryptographic standards (Barker and Kelsey, 2012; Pornin, 2013).

After obtaining the permutation index list $\mathbb{P}$, we proceed with the detection and extraction of the hints from I to assemble Π. This is done by the function HINTS_DETECTION$(\cdot)$ that takes as input I, and $\mathbb{P}$ (see line 3). Notice that for the synchronization to be successful, the same permutation had to be used to embed

Input: k, I, η
1 `seed` $\leftarrow$ PRNG(k)
2 $\mathbb{P} \leftarrow$ KEYED_PERMUTATION(seed,$\lfloor |I|/2 \rfloor$)
3 $\Pi \leftarrow$ HINTS_DETECTION$(I, \mathbb{P})$
4 $\zeta_2 \leftarrow$ CODE_BUILDER$(\Pi, \eta, k, \mathbb{P})$
5 $\zeta_6 \leftarrow$ BASE_CONVERT$(\zeta_2, 2 \rightarrow 6)$
6 $N_\emptyset \leftarrow$ ENCODE_WATERMARK(ζ_6)

Algorithm 1. Watermark encoding procedure.

Input: I, $\mathbb{P}$
Output: Π

1 $\Pi[\lfloor |I|/2 \rfloor] = 0$
2 **foreach** $p \in \mathbb{P}$ **do**
3 $\quad x \leftarrow I[2p]$
4 $\quad y \leftarrow I[2p+1]$
5 $\quad b_p \leftarrow (x - y) \bmod 2$
6 $\quad$ set (p, b_p) to Π
7 **return** Π

Algorithm 2. HINTS_DETECTION FUNCTION.

the hints into I. Nevertheless, as previously mentioned, as long as KEYED_PERMUTATION(·) is used with the same parameter values, this is guaranteed.

Algorithm 2 breaks down the logic of HINTS_DETECTION(·). It takes as input the parameters described above and returns an ordered hint set as Π. The operations depicted in this algorithm correspond to the extraction of bits based on the difference expansion method. Therefore, an even number of items is required in I. Thus, if an odd number of values is contained in I, one parameter is not involved in the synchronization.

First, Π is allocated as an array of $\lfloor |I|/2 \rfloor$ bits with all entries initialized to 0 (see line 1). The number of elements in Π is defined by $|I|$, which denotes the cardinality of I. The symbols $\lfloor \cdot \rfloor$ denote the floor function, used in case I is composed of an odd number of elements. $\mathbb{P}$ is utilized to partition I into adjacent pairs $(x, y) = (I_{2p}, I_{2p+1})$ (see lines 3 and 4). Scanning pairs in this order, we read the bit embedded by difference expansion and store it into b_p (see line 5).[7] Finally, the bit b_p obtained for the current pair of values is added to the position indexed by p in Π (see line 6).

After obtaining Π, line 4 of Algorithm 1 computes ζ_2 using the function CODE_BUILDER(·) that takes as input Π, η, k and $\mathbb{P}$. Considering this approach is symmetric, the key supplied to CODE_BUILDER(·) must match the key used when encoding the watermark hints in I. Algorithm 3 breaks down CODE_BUILDER(·), which assembles its output by mapping each index in $\mathbb{P}$ to a target position in ζ_2 via a keyed, uniform index function, defined as KEYED_INDEX(·) (see lines 5 to 7). First, the binary arrays ζ_2, `ones`, and `zeros`, each of length η, are initialized with 0 (see lines 1 to 3). The arrays `ones` and `zeros` are used to check if, for the same position in ζ_2, different bit values are identified in Π, which basically means data tampering is being performed in I. The bit targeted in Π can be obtained after knowing its index, which is extracted from $\mathbb{P}$ (see lines 5 and 7). Depending on the value of the bit selected from the p−th position in Π, the variables `ones` and `zeros` (counting the number of bits with 1 and 0 values for the matching position j, respectively) are incremented by one (see lines 8 to 11). Once all indices from $\mathbb{P}$ are considered, the consistency checking of the values detected for each position in ζ_2 proceeds (see lines 12 to 21).

7 More details about the way difference expansion works are given in Tian (2003) and Gupta and Pieprzyk (2009).

Input: Π, η, k, $\mathbb{P}$
Output: ζ_2

1 $\zeta_2[\eta] \leftarrow 0$
2 `ones`$[\eta] \leftarrow 0$
3 `zeros`$[\eta] \leftarrow 0$
4 `BI_LAB` $\leftarrow$ "KBit|v1"
5 **for** $i \leftarrow 0$ **to** $|\mathbb{P}|-1$ **do**
6 $\quad p \leftarrow \mathbb{P}[i]$
7 $\quad j \leftarrow$ KEYED_INDEX(p, η, k)
8 $\quad$ **if** $\Pi[p] = 1$ **then**
9 $\quad\quad$ `ones`$[j] \leftarrow$ `ones`$[j]+1$
10 $\quad$ **else**
11 $\quad\quad$ `zeros`$[j] \leftarrow$ `zeros`$[j]+1$
12 **for** $j \leftarrow 0$ **to** $\eta-1$ **do**
13 $\quad o \leftarrow$ `ones`$[j]$
14 $\quad z \leftarrow$ `zeros`$[j]$
15 $\quad$ **if** $(o > 0)$ *and* $(z = 0)$ **then**
16 $\quad\quad \zeta_2[j] \leftarrow 1$
17 $\quad$ **else if** $(z > 0)$ *and* $(o = 0)$ **then**
18 $\quad\quad \zeta_2[j] \leftarrow 0$
19 $\quad$ **else**
20 $\quad\quad$ `msg` $\leftarrow$ `BI_LAB` $\|$ "|chaos|j=" $\| j \|$ "|o=" $\| o \|$ "|z=" $\| z$
21 $\quad\quad \zeta_2[j] \leftarrow$ KEYED_BIT(K,msg) // discrepancy $\Rightarrow$ keyed chaos
22 **return** ζ_2

Algorithm 3. CODE_BUILDER FUNCTION.

Consistency at position j is established by inspecting the tallies in `ones` and `zeros` for that index. If `ones`$[j] > 0$ and `zeros`$[j] = 0$, we set $\zeta_2[j] \leftarrow 1$ (see lines 15 and 16); if `zeros`$[j] > 0$ and `ones`$[j] = 0$, we set $\zeta_2[j] \leftarrow 0$ (see lines 17 and 18). Otherwise, either no votes (`ones`$[j] =$ `zeros`$[j] = 0$) or a discrepancy (`ones`$[j] > 0 \wedge$ `zeros`$[j] > 0$), we treat the event as potential tampering and resolve it with a keyed tie-breaker KEYED_BIT(·) (see lines 20 and 21). This yields a deterministic, key-scoped bit that prevents chance agreement on manipulated inputs while remaining reproducible during verification. The generative key-based routines KEYED_PERMUTATION(·), KEYED_INDEX(·), and KEYED_BIT(·) are described in Section 4.5.

After the function CODE_BUILDER returns ζ_2, the code is represented in base-6 notation (denoted by ζ_6) by a simple base change operation performed by the function BASE_CONVERT(·) (see line 5 of Algorithm 1). Next, the graph representing the watermark is generated by the function ENCODE_WATERMARK(·) taking as input ζ_6 (see line 6).

The generation of the watermark is based on the graph representation of base-6 numbers presented in Collberg and Thomborson (1999). Algorithm 4 details the construction of the graph G from ζ_6. For this, ζ_6 is treated as an array of μ elements. The algorithm receives ζ_6 (which can also be used to obtain μ), and returns the head node N_0 of a circular list representing the graph. The watermark construction works by building a circular

Input: ζ_6, $\mu \leftarrow |\zeta_6|$
Output: N_0

1 $N_r \leftarrow$ Node(next, digit), $\forall r \in \{0, 1, \ldots, \mu-1\}$
2 **for** $r \leftarrow 0$ **to** $\mu - 1$ **do**
3 $\quad N_r$.next $\leftarrow N_{(r+1) \bmod \mu}$
4 **for** $r \leftarrow 0$ **to** $\mu - 1$ **do**
5 $\quad$ **if** $\zeta_6[r] = 0$ **then**
6 $\quad\quad N_r$.digit $\leftarrow$ null
7 $\quad$ **else**
8 $\quad\quad t \leftarrow (r + \zeta_6[r] - 1) \bmod \mu$
9 $\quad\quad N_r$.digit $\leftarrow N_t$
10 **return** N_0

Algorithm 4. ENCODE_WATERMARK FUNCTION.

singly linked list of μ nodes $N_0, \ldots, N_{\mu-1}$, where each node has two pointer fields: next and digit. The next field forms a ring, according to N_r.next $= N_{(r+1) \bmod \mu}$ with $r \in \mathbb{Z}^+ | 0 \leq r \leq \mu - 1$. The watermark is encoded solely by the digit pointers: if $\zeta_6[r] = 0$ then N_r.digit $=$ null; otherwise N_r.digit $= N_{(r+\zeta_6[r]-1) \bmod \mu}$. Intuitively, the digit is the forward offset along the ring: 1 = self, 2 = next, 3 = next–next, and so on (with modulo wrap-around); 0 is indicated by the absence of a pointer.

In the algorithm, all nodes are allocated with fields next, digit (see line 1). Next, the circular link between the nodes is created (see lines 2 and 3). Then, the values of the digit pointers are computed and assigned to each node depending on each value stored in each position of ζ_6, according to the criteria previously described (see lines 4 to 9). Finally, the head node of the structure representing the watermark is returned (see line 10).

The entire watermark graph can be traversed from any head node. Given access to the first node, the decoder assumes the structure matches the circular list described above. Algorithm 5 describes the function DECODE_WATERMARK(·), used for the decodification. It takes as input a head node, denoted as N'_0 (assumed to be the entry of the graph) and the length μ of the code expected to be extracted from the identified watermark. First, the function creates and initializes ζ_6 as an array of μ items (see line 1). Then, it sets $N_0 \leftarrow N'_0$ (see line 2). Next, it proceeds to create the circular list used to represent the graph. This is done by following next pointers for μ steps to collect the nodes in order from N_0 to $N_{\mu-1}$ (see lines 3 and 4). It is important to notice that the selection of N'_0 only rotates the sequence and thus does not affect the recovered digits.

Next, the algorithm proceeds to decode each digit locally. For the position r, if N_r.digit $=$ null then $\zeta_6[r] \leftarrow 0$ (see lines 6 and 7). Otherwise, it starts at N_r and advances along next, counting steps s until the digit target is reached, and returns $\zeta_6[r] \leftarrow s+1$ (so 1 = self, 2 = next, 3 = next-next, etc.) (see lines 9 to 17). The algorithm also uses a guard $s > \mu$ that flags an invalid structure (e.g., broken ring or unreachable digit) and prevents non-termination (see lines 15 and 16). Finally, after considering all nodes, if no error is reported, the recovered ζ_6 is returned (see line 18).

Input: N'_0, μ
Output: ζ_6

1 $\zeta_6[\mu] \leftarrow 0$
2 $N_0 \leftarrow N'_0$
3 **for** $t \leftarrow 1$ **to** $\mu-1$ **do**
4 $\quad N_t \leftarrow N_{t-1}$.next
5 **for** $r \leftarrow 0$ **to** $\mu-1$ **do**
6 $\quad$ **if** N_r.digit $=$ null **then**
7 $\quad\quad \zeta_6[r] \leftarrow 0$
8 $\quad$ **else**
9 $\quad\quad s \leftarrow 0$
10 $\quad\quad \hat{N} \leftarrow N_r$
11 $\quad\quad$ **repeat**
12 $\quad\quad\quad \hat{N} \leftarrow \hat{N}$.next
13 $\quad\quad\quad s \leftarrow s+1$
14 $\quad\quad$ **until** $\hat{N} = N_r$.digit or $s > \mu$
15 $\quad\quad$ **if** $s > \mu$ **then**
16 $\quad\quad\quad$ **error** "invalid structure (unreachable digit)"
17 $\quad\quad \zeta_6[r] \leftarrow s+1$
18 **return** ζ_6

Algorithm 5. DECODE_WATERMARK FUNCTION.

Algorithm 6 defines the process executed by the *Controller*. It takes as input the assumed initial node N'_0 from which the watermark is accessed, and the code ζ, which is used for comparison in assessing the authenticity of the parameters. According to the methods previously described to build Π from I, ζ is obtained and passed to the *Controller* based on blind principles, without requiring the original set of parameters, which would increase the risk of exposure of the protected content, offering unnecessary opportunities to malicious actors of intercept the data and compromise our approach's performance, and therefore, the security it provides. Nevertheless, ζ can be obtained from P or a second program deployed in the environment. Static watermarking techniques can be used to store ζ in P if required.

As depicted in Algorithm 6, the *Controller* starts by converting ζ into ζ_6 in order to obtain μ, which is required to build the code from the watermark (see lines 1 and 2). Next, using the function DECODE_WATERMARK(·), it proceeds to decode the watermark from the node N'_0 (see line 3). Then, it converts $\hat{\zeta_6}$ into $\hat{\zeta}$, which is the alleged secret code in decimal notation (see line 4). Finally, if $\zeta = \hat{\zeta}$, the authenticity of the parameters is verified and the process informs P that there is no risk in executing F with I (as previously defined, F^I) (see lines 5 and 6). On the other hand, if $\zeta \neq \hat{\zeta}$, the alternative flow is invoked to address the lack of authenticity in the parameters. Therefore, the *Controller* instruct P for the execution of F^ϕ instead (see lines 7 and 8). Calling F^ϕ does not necessarily mean the execution of the function. It can also refer to the execution of an alternative logic in P. Since the *Controller* instructs P and does not pass parameters to it directly, I is not considered among the inputs of Algorithm 6.

Input: N'_0, ζ

1 $\zeta_6 \leftarrow$ BASE_CONVERT(ζ, 10→6)
2 $\mu \leftarrow |\zeta_6|$
3 $\hat{\zeta}_6 \leftarrow$ DECODE_WATERMARK(N'_0, μ)
4 $\hat{\zeta} \leftarrow$ BASE_CONVERT($\hat{\zeta}_6$, 6→10)
5 **if** $\zeta = \hat{\zeta}$ **then**
6 RESUME P CALL TO F WITH I
7 **else**
8 RESUME P CALL USING ALTERNATIVE FLOW

Algorithm 6. *Controller* procedure.

It is important to note that watermark encoding and detection occur during program execution. Furthermore, the combination of static and dynamic approaches can increase resilience to our proposal. The parameter authentication approach has some features that contribute to its success. The input is never directly compared to a plain value in P. No visible hash or key is checked; only graph logic is used. Operations such as reverse engineering become more challenging, especially when the node structure is obfuscated and the watermark construction is triggered only under certain conditions.

This solution can be applied to tamper detection, where any modification of the input parameters renders the detected watermark invalid. Furthermore, since cryptographic hash functions are computationally efficient, this approach requires minimal computational resources. Finally, metadata for additional validation, such as timestamps or user IDs, can be considered when building W, thereby increasing the meaningfulness and robustness of the proposal.

4.5 Applied generative key-based functions

Three critical functions are essential for the correct functioning of the parameter authentication approach. They are KEYED_PERMUTATION(·), KEYED_INDEX(·), and KEYED_BIT(·). Although their goal is briefly mentioned when presenting the approach, this section describes them in more detail.

4.5.1 The KEYED_PERMUTATION function

The first generative key-based function, KEYED_PERMUTATION(·), is used to synchronize the watermark hints in I. It takes as input a `seed` and an even integer $n = \lfloor |I|/2 \rfloor$, and produces a keyed permutation of index pairs in I for the difference expansion. The output is the deterministic permutation index list $\mathbb{P}$, as previously formalized. Its essential properties are reproducibility for identical parameters, unpredictability without the seed, and consistency with PRF-driven derivations.

Algorithm 7 specifies the procedure for KEYED_PERMUTATION(·). Its inputs are a secret `seed`, a pair-count n, and an optional context string `ctx` for domain separation. If `ctx` is omitted, a default value is used. The function initializes $\mathbb{P}$ as the identity ordering $[0, 1, \ldots, n-1]$ (line 1) and

Input: `seed`, n, `ctx` ← "KPerm|v1"
Output: $\mathbb{P}$
1 $\mathbb{P} \leftarrow [0, 1, \ldots, n-1]$
2 **for** $i \leftarrow n-1$ **to** 1 **do**
3 label ← "perm|" ‖ ctx ‖ "|i=" ‖ i ‖ "|c=" ‖ c
4 dig ← HMAC-SHA256(seed, label)
5 $r \leftarrow$ first 64 bits of dig
6 $m \leftarrow i+1$
7 $L \leftarrow 2^{64} - (2^{64} \bmod m)$
8 **while** $r \geq L$ **do**
9 $c \leftarrow c+1$
10 recompute r as above
11 rejection sampling $j \leftarrow r \bmod m$
12 swap $\mathbb{P}[i], \mathbb{P}[j]$
13 $c \leftarrow c+1$
14 **return** $\mathbb{P}$

Algorithm 7. KEYED_PERMUTATION FUNCTION

applies the Durstenfeld/Knuth Fisher-Yates shuffle from $i = n-1$ down to 1 (line 2). At each iteration, it derives a swap index $j \in 0, \ldots, i$ via a keyed PRF. Specifically, it builds a domain-separated label from `ctx` and loop metadata (i, c) (see line 3), computes a hash-based message authentication code (HMAC) using the cryptographic hash function SHA-256 (see line 4), extracts the first 64 bits as an integer r (see line 5), and maps r uniformly to $[0, i]$ via rejection sampling with modulus $m = i+1$ and cutoff $L = 2^{64} - (2^{64} \bmod m)$ (see lines 6 and 7). If $r \geq L$, the counter c is incremented and r re-derived until valid (see lines 8 to 10). The swap index is then set as $j = r \bmod m$ (line 11), and $\mathbb{P}[i]$ is exchanged with $\mathbb{P}[j]$ (see line 12). The counter c advances once per iteration to decorrelate labels (see line 13).

Under standard assumptions, HMAC behaves as a secure PRF (Bellare et al., 1996; Krawczyk et al., 1997), making the derived swap indices computationally indistinguishable from independent uniform draws. Since the Fisher-Yates shuffle with uniform choices produces the uniform distribution over all $n!$ permutations (Durstenfeld, 1964; Knuth, 1997), this construction yields an unbiased, key-dependent permutation. The use of a context string for domain separation follows established practice in the HMAC-based Key Derivation Function (HKDF), where the `info` field binds outputs to application-specific context (Krawczyk and Eronen, 2010). Finally, rejection sampling guarantees provably uniform bounded integers with efficient implementation (Lemire, 2019).

4.5.2 The KEYED_INDEX function

Another critical function in this work is KEYED_INDEX(·), invoked by CODE_BUILDER(·) to determine the target index in ζ_2 for a given position p in $\mathbb{P}$, the length η of ζ_2, and the secret key k (see line 7 of Algorithm 3). Algorithm 8 specifies its procedure. The function deterministically maps a host index p to a uniform position `idx` $\in [0, \eta)$ using k. Assuming HMAC is a secure PRF, the output is reproducible for identical inputs yet computationally

Input: p, η, k
Output: idx

1 limit $\leftarrow 2^{64} - (2^{64} \bmod \eta)$
2 $c \leftarrow 0$
3 BI_LAB $\leftarrow$ "KInd|v1"
4 **repeat**
5 msg $\leftarrow$ BI_LAB $\|$ "−" $\|$ p $\|$ "−" $\|$ c
6 $D_{p,c} \leftarrow$ HMAC−SHA256(k, msg)
7 $r \leftarrow$ first 64 bits of $D_{p,c}$
8 $c \leftarrow c+1$
9 **until** $r < limit$
10 **return** idx $\leftarrow (r \bmod \eta)$

Algorithm 8. KEYED_INDEX FUNCTION.

indistinguishable from a uniform draw without knowledge of k (Bellare et al., 1996; Krawczyk et al., 1997). Uniformity is enforced via rejection sampling to eliminate modulo bias when reducing a 64-bit value to a smaller range (Lemire, 2019). Domain separation is achieved by combining a fixed label (BI_LAB), p, and a counter c into the HMAC input, in the spirit of HKDF's info field (Krawczyk and Eronen, 2010). This primitive supports keyed scan orders, ring-head selection, and decoy placement, where indices must be reproducible but unpredictable.

In detail, Algorithm 8 begins by computing the cutoff limit $= 2^{64} - (2^{64} \bmod \eta)$ (see line 1), ensuring unbiased indices through rejection sampling, as also applied in KEYED_PERMUTATION(·). It then initializes the counter c and the fixed domain label BI_LAB (see lines 2 and 3). Each iteration constructs a message msg, computes an HMAC–SHA256 digest $D_{p,c}$ using msg and k, and interprets the first 64 bits as an integer r (see lines 5 to 7). The counter c is incremented until a valid $r <$ limit is obtained (see lines 8 and 9). Finally, the algorithm outputs idx $\leftarrow r \bmod \eta$ (see line 10). In this way, each position p in a permutation is deterministically associated with a particular index in ζ_2.

4.5.3 The KEYED_BIT function

The final keyed PRF in this work is KEYED_BIT(·), which produces a deterministic pseudorandom bit $b \in \{0, 1\}$ from a domain-separated label. For any fixed pair of parameters (k, label) this function always returns the same bit, ensuring reproducibility. On the other hand, to any party without k, the output is computationally indistinguishable from random under the standard assumption that HMAC is a secure PRF (Bellare et al., 1996; Krawczyk et al., 1997). The use of a structured label (e.g., ctx|purpose|...) achieves domain separation, analogous to the info field in HKDF (Krawczyk and Eronen, 2010), so bits used in different roles do not collide. This primitive is well-suited for keyed tie-breaking, chaos-inducing decisions, or any context where a minimal yet reproducible source of unpredictability is needed.

In this work, this function is responsible for adding chaos to the reconstructed code ζ_2 when the slightest inconsistency is detected in Π when executing CODE_BUILDER(·) (see lines 19 to 21 of Algorithm 3). This ensures that ζ_2 diverges from its authentic value, preventing accidental reconstruction by sheer chance and thereby enabling reliable detection of parameter tampering.

Input: k, label
Output: $b \in \{0, 1\}$

1 msg $\leftarrow$ UTF8(label)
2 dig $\leftarrow$ HMAC−SHA256(k, msg)
3 $r \leftarrow$ first 8 bytes of dig interpreted as a 64-bit unsigned integer
4 $b \leftarrow r \bmod 2$
5 **return** b

Algorithm 9. KEYED_BIT FUNCTION

Algorithm 9 specifies the implementation of KEYED_BIT(·). It first encodes the domain-separated label into bytes (UTF-8) (see line 1) and computes an HMAC–SHA256 digest with k over that byte string (see line 2). Then, it interprets the first 64 bits of the digest as an unsigned integer r (see line 3) and outputs $b \leftarrow r \bmod 2$ (see line 4). Since 2 divides 2^{64} evenly, this reduction is unbiased and requires no rejection sampling. Finally, the algorithm returns b (see line 5). For generating wider indices in $[0, m)$ with arbitrary m, a companion routine employing rejection sampling must be used to avoid modulo bias.

5 Experimental results

This section presents the results that validate the applicability of the proposed solution. The evaluation focuses on performance in detecting tampered input parameters and on preserving function outcomes when parameter authenticity is confirmed. The experiments also assess the proposal's impact on the protected code, considering both the modifications required for integration and their effect on performance. Specifically, results are reported for code size and execution time, showing effective detection of tampered inputs across different values of I. Moreover, the approach requires neither significant resources nor a complex environment for development and execution.

5.1 Experimental setup

We start from the basis that dynamic watermarking techniques have not been applied for this goal before. We selected Python 3.12.7 for implementing our approach, considering its extensive use in developing solutions that address the challenges we mentioned earlier. The development environment was Spyder 5.5.1 Integrated Development Environment (IDE) from the Anaconda Navigator 2.6.4 Python distribution. The runtime environment consisted of a PC equipped with an Intel i7-7700K processor, clocked at 4.20 GHz, 32.0 GB of RAM, and running Windows 10 (AMD64) Pro OS.

We selected five Python functions implementing different functionalities to validate our approach. Each function takes an array of integers as input, but the acceptable range of values

depends on the specific logic implemented. A brief description of each function is provided below:

- F_A: Function defined as `total_revenue_cents(·)` which takes a list of transaction amounts expressed in cents and sums them. It returns a single integer, which is the total revenue in cents, with 0 for an empty list. Negative values (e.g., refunds) are naturally included in the sum, so the result reflects net revenue.
- F_B: Function defined as `median_basket_size(·)` which receives a list of item counts per order and returns the median basket size as an integer. For an even number of orders, it uses the floor of the average of the two middle values; for an empty list, it returns 0. It is useful for tracking typical cart size while being robust to extreme outliers.
- F_C: Function defined as `sla_breach_rate(·)` which takes response times in milliseconds and returns the fraction of requests that exceed the Service Level Agreement (SLA) threshold. By default, the threshold is 2000 ms (tunable), and an empty list yields 0.0. The result is a float in $[0, 1]$, representing the share of breaches.
- F_D: Function defined as `churn_flags(·)` which takes, for each customer, the number of days since last activity and returns a boolean list flagging likely churners. Any value greater than or equal to the inactivity window (default 30 days) is flagged `true`. The output length matches the input. An empty input returns an empty output.
- F_E: Function defined as `staffing_alert_levels(·)` which takes a time-ordered list of tickets per hour and classifies each hour as "normal", "high", or "critical". By default: $\leq 15 \rightarrow$ normal, 16 to 30 $\rightarrow$ high, $> 30 \rightarrow$ critical; the thresholds are easy to adjust. It returns a list of labels aligned 1:1 with the input hours.

For each function, we use five different sets of parameters, each consisting of an array of 32 integers. Each parameter is identified by a name that links it to the function on which it is used. The structure of the set name is I_{F-D} where $F \in \{A, B, C, D, E\}$ identifies the function and $D \in \{1, 2, 3\}$ denotes the parameter set for that function. According to this, I_{A-3} denotes the third set of parameters for the function F_A (a.k.a `total_revenue_cents(·)`.

Table 2 summarizes the characteristics of each dataset used as input for each function. It contributes to understanding the range of values that each function handles and the differences across the cases analyzed. Specifically, for each array, the table reports the maximum value (column **MAX**), the minimum value (column **MIN**), the average (column **AVG**), and the standard deviation (column **STD_DEV**). These statistics provide insight into the variability and distribution of values within each dataset.

5.2 Detection of data tampering

The validation of the proposed approach is guided by the research questions introduced earlier. This section presents the results that address the first question: *"Does the proposed approach detect tampering of input parameters with a high true-positive rate and low false-positive rate across representative workloads?"* To answer this, it is first necessary to define: (i) how to compute the true-positive rate (TPR) and what qualifies as a high value, (ii) how to compute the false-positive rate (FPR) and what qualifies as a low value, and (iii) which representative workloads are suitable for validating the proposal.

The true-positive rate (TPR) measures the proportion of tampered inputs that are correctly detected as tampered. In this work, it is computed according to Equation 1, where TP denotes the number of tampered cases correctly identified and FN the number of tampered cases missed (Fawcett, 2006; Sokolova and Lapalme, 2009). In our context, a high TPR indicates near-exhaustive detection of tampering and should be defined a priori. We formally adopt TPR $\geq$ 0.99 as the threshold for a high TPR, given the importance of ensuring the proper functioning of the proposed approach.

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{1}$$

The false-positive rate (FPR) measures the proportion of authentic inputs that are incorrectly flagged as tampered. In this work, it is computed according to Equation 2, where FP denotes the number of authentic inputs incorrectly classified as tampered and TN the number of authentic inputs correctly classified as not tampered (Fawcett, 2006; Sokolova and Lapalme, 2009). In our context, a low FPR indicates rare false alarms and should be defined a priori. We formally adopt FPR $\leq$ 0.01 as the threshold for a low FPR.

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} \tag{2}$$

Finally, we define a representative workload as a curated set of inputs and execution conditions that faithfully reflect the system's intended use. Such a workload should preserve the statistical properties of the data, the shape and scale of inputs, and the mix of operations exercised by the application. It should also include both benign variations (e.g., reordering, padding) and adversarial perturbations consistent with the threat model, all under reproducible settings.

In our evaluation, the five previously introduced functions constitute the representative workload. Together they span order-insensitive *aggregation* (as in F_A), a robust *order statistic* (as in F_B), *rate/threshold* computation near decision boundaries (as in as in F_C), vectorized *boolean flagging* (as in F_D), and *multi-level classification* on time series (as in F_E). This set covers both order-insensitive and order-sensitive behaviors, single-output and vector outputs, and varying sensitivities to bit-level edits and structured tampering, while also admitting benign controls (e.g., pure reordering) to estimate the FPR. Instantiated across multiple scales and data distributions, these functions provide a realistic and reproducible proxy for the target application domain, thereby satisfying the representativeness required by the first research question.

To evaluate the performance of our proposal in detecting data tampering, we conducted 15 experiments per function. Each function was tested with its corresponding datasets, applying five

TABLE 2 Datasets used for the validation of the approach.

F	I	Description	MAX	MIN	AVG	STD_DEV
F_A	$I_{A\text{-}1}$	Subscriptions with occasional refunds (negatives)	4999	−999	1217.94	1131.15
	$I_{A\text{-}2}$	Enterprise invoices (large amounts in cents)	210000	76000	133421.88	36019.77
	$I_{A\text{-}3}$	Microtransactions + freebies (zeros)	199	0	111.69	69.29
F_B	$I_{B\text{-}1}$	Even-sized sample with typical small baskets	4	1	2.19	0.88
	$I_{B\text{-}2}$	Many zeros (browsers/no-purchase) mixed with small buys	2	0	0.5	0.75
	$I_{B\text{-}3}$	Wholesale / large carts	22	8	14.22	3.81
F_C	$I_{C\text{-}1}$	Mixed near-threshold with some spikes	4000	1600	2189.84	529.07
	$I_{C\text{-}2}$	All fast (no breaches)	1500	120	764.69	321.62
	$I_{C\text{-}3}$	All slow (incident/major degradation)	6100	2800	4589.06	924.47
F_D	$I_{D\text{-}1}$	Mixed recency across users	90	0	24.63	20.01
	$I_{D\text{-}2}$	Mostly active users (all below threshold)	20	1	8.38	4.68
	$I_{D\text{-}3}$	Dormant population (all flagged)	120	30	58.81	23.17
F_E	$I_{E\text{-}1}$	Two-day hourly pattern, steady with small bumps	15	5	9.16	2.97
	$I_{E\text{-}2}$	Midday surge then cool-down over two days	30	4	15.38	7.54
	$I_{E\text{-}3}$	Incident window with sustained critical load	50	7	23.88	13.08

different degrees of data modification. We deliberately selected the smallest possible number of elements and applied minimal changes to stress-test our approach, assessing whether tampering could still be detected even when the modifications were so slight that they sometimes did not alter the function's output. Tampering was introduced by pseudo-randomly selecting between one and five input values and modifying the least significant bit (*lsb*) of each chosen value.

The results are presented in Table 3 for each function (column F) and its associated dataset (column I). They are organized into four groups of columns: **Tampering Detected**, **TPR**, **FPR**, and **Output Modified**. Binary outcomes are denoted as ✓ (yes) and ✗ (no). Within each group, five columns correspond to the number of array items pseudo-randomly selected for *lsb* modification. The Output Modified column illustrates that tampering does not always lead to changes in the function result. Nevertheless, detecting such tampering remains essential to uncover potential vulnerabilities, prevent unauthorized access or data leaks, and mitigate the risk of future output changes if function logic is modified under approved requirements.

As shown in the table, tampering was successfully detected in every case, thereby fulfilling the requirement established by the first research question: demonstrating that the approach achieves **TPR** $= 1$ and **FPR** $= 0$. Detection was achieved even in cases where the modifications did not affect the output that the function would have produced on untampered data. This highlights another strength of our approach. Nevertheless, it is important to note that this behavior depends heavily on the function's requirements and implementation. For example, the output of F_A was always different, since it performs a summation of all input values. By contrast, F_B, which computes the average of the input values and returns a rounded result, is less likely to change under small modifications. Finally, changes in the outputs of F_D and F_E were rarely detected, as their results only vary when tampering affects values close to the classification boundaries defined by the functions.

5.3 Output accuracy variations

The results presented in this section correspond to the experiments conducted to address the second research question: *"Does integrating the approach preserve the functional output of protected functions when the input parameters are authentic?"* To answer this, we obtained the outputs of each function using the various datasets and compared them with the outputs obtained after confirming that no tampering occurred and the input parameters were restored. Success is defined as confirming that the outputs remain unchanged. Additionally, we provide the outputs that each function would produce if the input parameters were not restored after decoding the watermark, allowing a comparison of the distortion introduced by retaining the watermark hints.

Table 4 shows the outcomes of functions F_A, F_B, and F_C, for each input set, both before and after applying our approach. The group labeled **After** contains two columns: **Encoded**, which refers to parameters still containing the watermark hints (not restored to their original state), and **Restored**, which contains the data after extracting the hints, restoring the parameters, and verifying the authenticity of the watermark. Values in the **Encoded** column include a number in parentheses representing the difference from the output obtained before embedding the watermark hints.

The table shows how distortion can result in different outputs. However, in some cases, no distortion occurs because increases in certain values are compensated by decreases in others during the embedding of the hints or due to the way the function processes the input. This behavior is observed in all cases of function F_B, for example. Additionally, the results from these experiments confirm

TABLE 3 Results of data tampering detection using the input parameters authentication approach ($\zeta_2 = 1011$, $k =$ "K82Sec").

F	I	Tampering detected					TPR					FPR					Output modified				
		1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
F_A	$I_{A\text{-}1}$	✓	✓	✓	✓	✓	1	1	1	1	1	0	0	0	0	0	✓	✓	✓	✓	✓
	$I_{A\text{-}2}$	✓	✓	✓	✓	✓	1	1	1	1	1	0	0	0	0	0	✓	✓	✓	✓	✓
	$I_{A\text{-}3}$	✓	✓	✓	✓	✓	1	1	1	1	1	0	0	0	0	0	✓	✓	✓	✓	✓
F_B	$I_{B\text{-}1}$	✓	✓	✓	✓	✓	1	1	1	1	1	0	0	0	0	0	✗	✗	✗	✗	✗
	$I_{B\text{-}2}$	✓	✓	✓	✓	✓	1	1	1	1	1	0	0	0	0	0	✗	✗	✗	✗	✗
	$I_{B\text{-}3}$	✓	✓	✓	✓	✓	1	1	1	1	1	0	0	0	0	0	✗	✗	✗	✗	✗
F_C	$I_{C\text{-}1}$	✓	✓	✓	✓	✓	1	1	1	1	1	0	0	0	0	0	✗	✗	✗	✗	✗
	$I_{C\text{-}2}$	✓	✓	✓	✓	✓	1	1	1	1	1	0	0	0	0	0	✗	✗	✗	✗	✗
	$I_{C\text{-}3}$	✓	✓	✓	✓	✓	1	1	1	1	1	0	0	0	0	0	✗	✗	✗	✗	✗
F_D	$I_{D\text{-}1}$	✓	✓	✓	✓	✓	1	1	1	1	1	0	0	0	0	0	✗	✗	✓	✓	✓
	$I_{D\text{-}2}$	✓	✓	✓	✓	✓	1	1	1	1	1	0	0	0	0	0	✗	✗	✗	✗	✗
	$I_{D\text{-}3}$	✓	✓	✓	✓	✓	1	1	1	1	1	0	0	0	0	0	✗	✗	✗	✗	✗
F_E	$I_{E\text{-}1}$	✓	✓	✓	✓	✓	1	1	1	1	1	0	0	0	0	0	✗	✗	✗	✗	✗
	$I_{E\text{-}2}$	✓	✓	✓	✓	✓	1	1	1	1	1	0	0	0	0	0	✗	✗	✗	✗	✗
	$I_{E\text{-}3}$	✓	✓	✓	✓	✓	1	1	1	1	1	0	0	0	0	0	✗	✗	✓	✓	✓

TABLE 4 Comparison of the outputs of functions F_A, F_B, and F_C before and after applying the input parameter authentication approach ($\zeta_2 = 1011$, $k =$ "K82Sec").

F	I	Before	After	
			Encoded	Restored
F_A	$I_{A\text{-}1}$	38, 974	38, 986 (+12)	38, 974
	$I_{A\text{-}2}$	4, 269, 500	4, 269, 512 (+12)	4, 269, 500
	$I_{A\text{-}3}$	3, 574	3, 580 (+6)	3, 574
F_B	$I_{B\text{-}1}$	2	2 (0)	2
	$I_{B\text{-}2}$	0	0 (0)	0
	$I_{B\text{-}3}$	14	14 (0)	14
F_C	$I_{C\text{-}1}$	0.53125	0.53125 (0)	0.53125
	$I_{C\text{-}2}$	0	0 (0)	0
	$I_{C\text{-}3}$	1	0.96875 (+0.03125)	1

TABLE 5 Comparison of the outputs of the function F_D before and after applying the input parameter authentication approach ($\zeta_2 = 1011$, $k =$ "K82Sec").

I	Before		After			
			Encoded		Restored	
	T	F	T	F	T	F
$I_{D\text{-}1}$	14	18	15 (+1)	17 (−1)	14	18
$I_{D\text{-}2}$	0	32	0 (0)	32 (0)	0	32
$I_{D\text{-}3}$	32	0	25 (−7)	7 (+7)	32	0

that our approach does not interfere with the outputs of the functions, as evidenced by the comparison between the **Before** and **Restored** columns.

Table 5 shows the results obtained when applying our approach to the function F_D with its corresponding inputs. Each output is presented in two columns, **T** and **F**, containing the number of `true` and `false` values in the boolean list returned by the function. Notice that in the Encoded column, the function's outputs sometimes remain unchanged. Additionally, because the output consists of a group of values, any distortion introduced by embedding the watermark hints is compensated across the different elements of the group (for example, under the **Encoded** column, values in **T** offset those in **F**). Finally, the fact that our approach does not interfere with the function's accuracy is evident from the matching values in the **Before** and **Restore** columns.

Table 6 shows the results of the experiments performed for F_E with its corresponding inputs. In this case, the function returns three integers, shown in columns **N**, **H**, and **C**, which stand for "normal", "high", and "critical", respectively. We can see that sometimes the encoding does not alter certain outputs (e.g., under the **Encoded** category, column **C** for dataset $I_{E\text{-}1}$, and column **H** for dataset $I_{E\text{-}2}$). Additionally, also in this case, it is clear that the distortion introduced by embedding the watermark hints is compensated across the different elements in each output. Finally, we can see that once the hints are extracted from I, the restored parameters allow the computation of the same results as those obtained with the original set of inputs.

As shown in the previous tables, we can confirm that our approach does not interfere with the outputs of the functions. This is ensured by the reversibility provided through difference expansion. When obtaining ζ, the set of input parameters is restored to its original state. If ζ does not match the expected

value, the watermark is not recognized, and an alternative flow is triggered, allowing P to take control and manage the process in a customized manner. Conversely, if the parameters are considered authentic, F is executed with authentic inputs, producing accurate results. This conclusion is supported by the fact that none of the experiments resulted in variations in the functions' outputs when the parameter authentication approach was applied.

5.4 Performance and overhead evaluation

This section addresses the third research question: *"Does integrating the approach impose a bounded, predictable overhead in code size and execution time, with runtime scaling linearly in input size across representative workloads?"* Here, we present the results of a set of experiments that evaluate the impact of our approach on the protected software. We report the effects on software size and how processing time is affected. These experiments allow us to quantify the overhead introduced by the watermarking mechanism and assess its practicality in real-world applications. By analyzing both memory footprint and execution time, we provide a comprehensive view of the trade-offs involved in applying our method.

5.4.1 Impact on code size

The measured overhead of our approach is small and predictable. Table 7 reports, for each program P wrapping a workload function, the source footprint before and after integrating the parameters authentication proposal. Columns **Size$_B$** and **Size$_W$** contains the total source size (in KiB) of the base program and its version with the watermarking engine integrated, **ΔSize** contains the absolute difference between them (which denotes a size increment), and **Overhead** is the percentage of the relative growth obtained according to Equation 3.

$$\text{Overhead} = \frac{\text{Size}_W - \text{Size}_B}{\text{Size}_B} \times 100 \tag{3}$$

The following columns, **SLOC$_B$** and **SLOC$_W$**, contain the number of lines of each source code, **Δ SLOC** presents their difference, and **Hook sites** count explicit instrumentation call sites (embed/verify) added in the program. The final record, **Total / Weighted avg.**, aggregates sizes and SLOC, and reports the size-weighted overhead percentage.

Across all five programs, integrating our approach produces a consistent and bounded footprint: code size rises by about 6 KiB per program, and SLOC increases are similarly stable (median **ΔSLOC** = 101, range 86–112). The largest percentage overhead is in `sla_app.py` (144.90%) and the smallest in `staffing_app.py` (111.65%), a pattern explained by fixed scaffolding added via two hook sites (embed/verify) per program. These results indicate a modular, predictable integration cost and support a positive deployment outlook, the parameters are restored losslessly prior to business logic, so authentic inputs preserve baseline outputs. Notice that the seemingly large percentages mainly reflect very small baselines, making the absolute increment negligible in typical services.

TABLE 6 Comparison of the outputs of the function F_E before and after applying the input parameter authentication approach ($\zeta_2 = 1{,}001$, $k =$ "K82Sec").

I	Before			After					
				Encoded			Restored		
	N	H	C	N	H	C	N	H	C
$I_{E\text{-}1}$	32	0	0	29 (−3)	3 (+3)	0 (0)	32	0	0
$I_{E\text{-}2}$	18	14	0	16 (−2)	14 (0)	2 (+2)	18	14	0
$I_{E\text{-}3}$	12	8	12	13 (+1)	9 (+1)	10 (−2)	12	8	12

5.4.2 Execution time variations

The solution proposed in this work does not have a significant impact on the execution time of each program P. Two main aspects of our approach may result in processing time changes for the tasks implemented in P: the size of the graph G used to represent the watermark and the number of elements in I containing watermark hints. We performed a set of experiments, noticing that these are the only two aspects that produce variations in P over time, but not in the same manner.

For the first experiments, we measured the time required to encode and decode the watermark hints from I while increasing the number of values in the input (denoted by $|I|$). For each input length, we randomly generated the values within the input domain and performed 300 trials per setting using the same watermark hints. This was done to reduce the standard deviation of the reported results. Figure 3 shows the results, which consists on the average execution time in microseconds.

We also conducted experiments to measure the time required to build and decode G while varying μ. Recall that μ denotes

TABLE 7 Source footprint per program before/after integrating the parameter authentication approach.

Program (F contained)	Size$_B$	Size$_W$	Δ Size	Overhead	SLOC$_B$	SLOC$_W$	Δ SLOC	Hook sites
`revenue_app.py` (F_A)	5.083008	11.097656	6.014648	118.33%	82	168	86	2
`basket_app.py` (F_B)	5.504883	11.929688	6.424805	116.71%	93	205	112	2
`sla_app.py` (F_C)	4.491211	10.999023	6.507812	144.90%	68	177	109	2
`churn_app.py` (F_D)	4.944336	11.014648	6.070312	122.77%	68	169	101	2
`staffing_app.py` (F_E)	5.381836	11.390625	6.008789	111.65%	83	183	100	2
Total / Weighted avg.	25.405274	56.431640	31.026366	122.13%	394	902	508	10

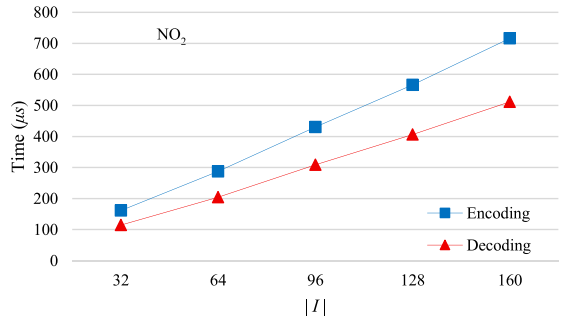

FIGURE 3
Encoding/decoding time variations of watermark hints as |I| increases.

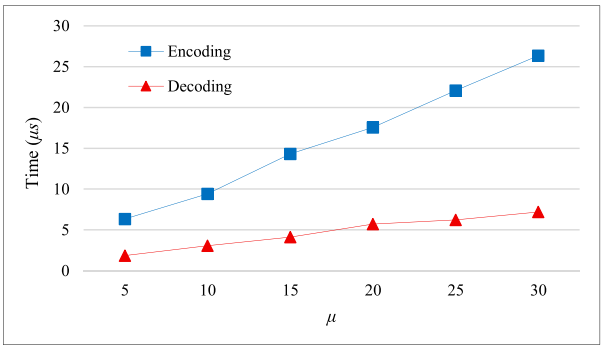

FIGURE 4
Build/decoding time of G as as μ increases.

the length of ζ_6 (see Algorithms 4, 5). For each selected value of μ, we generated a random instance of ζ_6, which was then used for encoding and decoding the watermark. For each setting, we performed 300 trials and reported the average time in microseconds. The results are shown in Figure 4.

As shown in Figure 3, reported times increase linearly with the length of I, confirming the expected $O(|I|)$ behavior of the approach for the engine responsible for encoding and decoding the watermark hints from I. We can see that decoding is consistently faster than encoding, because it performs a read-restore pass and a small, fixed rebuild, while encoding also allocates and writes modified pairs. The near-constant μs/element slopes and the stable encode-decode gap indicate predictable performance and straightforward capacity planning. Overall, the results support the practicality of the approach for runtime parameter authentication: complexity is linear in input size, overhead is stable, and the implementation scales smoothly to large arrays without surprises.

In terms of the time required for encoding and decoding the watermark, Figure 4 depicts how both times grow linearly with μ, confirming the expected behavior. Decoding is consistently faster because it performs one ring traversal to list nodes plus a constant-step local check per node, whereas encoding allocates the graph nodes and writes two pointers per node. The slight increase in the encoding slope is plausibly due to object allocation, garbage collection, and cache effects; decoding remains near $\sim$ 0.24-0.26 μs/node. Overall, the timing profile indicates that watermark graph construction and recovery are fast (sub-millisecond to a few milliseconds for thousands of nodes) and scale predictably, making the approach practical for runtime parameter authentication.

5.4.3 Memory overhead analysis

Although the proposed approach relies on the dynamic construction of data structures to encode and verify the watermark during execution, the additional memory footprint introduced by these operations was found to be negligible across all evaluated scenarios. In our experiments, each function received an input array of 32 integers, resulting in watermark structures with a size proportional to the binary representation of the secret code and the number of embedded hints. Even under these conditions, peak memory usage increased by less than 1.5% compared with baseline executions without watermarking. Moreover, because the watermark graph is created only during the parameter verification phase and is released immediately afterward, its lifetime in memory is short and its space complexity scales linearly with the size of the input parameter set. These results confirm that the proposed technique introduces no significant memory overhead and remains suitable for deployment in resource-constrained environments.

5.5 Approach robustness to code transformations

We evaluated the robustness of our approach to code transformations by considering the last research question: "*Are authentication decisions robust to compiler/interpreter optimizations, code obfuscation, common refactoring, and heterogeneous runtime environments?*" First, we checked whether the proposed approach is invariant under a simple interpreter optimization pass by running each program in three modes: default (`python`), optimized (`python -O`), and docstring-stripping optimized (`python -OO`). For each mode, and for each workload program, we execute the pipeline with identical inputs, comparing the results on tampering detection, and encoding/decoding times. The expectation is functional invariance (identical decisions and recovered codes) and only minor timing drift due to bytecode differences.

Figures 5–8 show that, under all three interpreter modes, the encoding and decoding curves retain the expected linear trend in the input size ($|I|$ for encoding the watermark hints, and μ for the watermark graph G). Optimization mostly produces modest timing shifts without changing qualitative behavior. The small deviations shown in the figures are consistent with bytecode changes, allocation and cache effects, and the removal of assertions/docstrings.

Crucially, optimization does not alter security outcomes. The tampering decisions are identical across all all three interpreter modes. Thus, the approach is robust to common compiler/interpreter optimizations. Its scalability ($O(|I|)/O(\mu)$), encode > decode gap, and correctness are preserved, while performance differences remain bounded and predictable. This

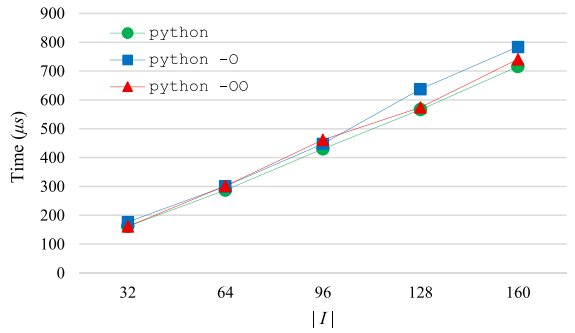

FIGURE 5
Consequences of code optimization in the time required for encoding the watermark hints as $|I|$ increases.

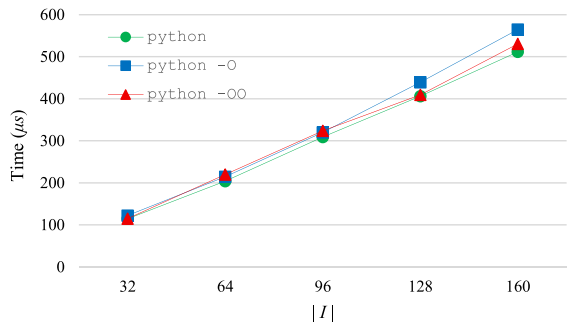

FIGURE 6
Consequences of code optimization in the time required for decoding the watermark hints as $|I|$ increases.

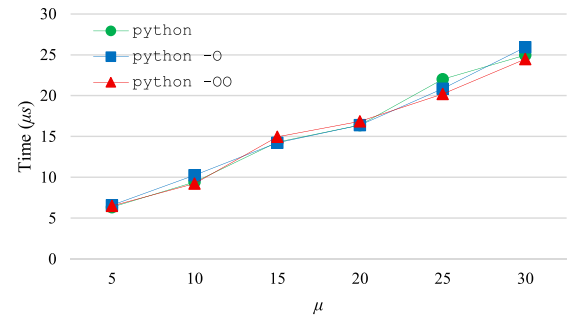

FIGURE 7
Consequences of code optimization in the time required for encoding G as μ increases.

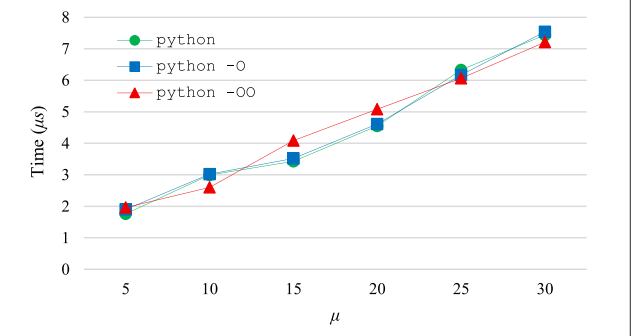

FIGURE 8
Consequences of code optimization in the time required for decoding G as μ increases.

supports deployability in optimized production configurations without weakening authentication guarantees.

We further assessed robustness to code-level transformations by applying: (i) lexical/minification changes (identifier renaming, removal of whitespace and docstrings), (ii) bytecode wrapping and encryption via a runtime loader, and (iii) native compilation and packaging to an executable. In parallel, we enacted common refactorings that preserve semantics, including function extraction/inlining, module reorganization, dead-code removal, and reorderings that do not affect data or control dependencies. For each transformed artifact, we ran the full pipeline with identical inputs, keys, and PRNG seeds, verifying tampering detection results, effects on the workload functions, and timing profiles. Across all obfuscation levels and refactorings, authentication decisions and recovered tags remained identical to the baseline, and the encode/decode time curves preserved the same linear trend with only bounded drift, mirroring the behavior observed under the `-O`/`-OO` optimization experiment.

5.6 Analysis of attack resilience and detection capabilities

In terms of robustness against the major categories of attacks described in Section 2.4, specifically subtractive, distortive, and additive, it is important to emphasize that the proposed approach follows the principles of a fragile watermarking technique. Its primary objective is not to remain undetectable under all possible modifications but rather to act as a sensitive safeguard against unauthorized operations. By detecting even minimal alterations to input parameters, the technique ensures that users are alerted whenever data authenticity may be compromised. Table 8 summarizes the overall resilience of the approach against these classical attack types.

Our solution demonstrates strong resistance to subtractive attacks because the correct watermark cannot be identified or reconstructed unless the precise input parameter values are used to achieve synchronization. Any attempt to remove or bypass the watermarking logic without providing the expected parameters will therefore fail, preventing the protected function from executing successfully.

This same principle underlies the approach's resilience to distortive attacks. Any modification to the software code or

TABLE 8 Effectiveness of the proposed approach against classical software watermarking attacks.

Attack	Resilience	Justification
Subtractive	High	Watermark reconstruction depends on correct parameter values; removal without synchronization is highly unlikely.
Distortive	High	Code or data modifications cause incorrect watermark generation, triggering authenticity alerts.
Additive	High	Reconstruction requires an exact bit match; any alteration leads to chaotic output and detection of tampering.

data aimed at undermining watermark detection will lead to the construction of an incorrect watermark, automatically triggering a suspicion of tampering. This behavior aligns with the core purpose of fragile watermarking, which is to flag even subtle unauthorized distortions as potential integrity violations.

Finally, the approach also provides strong protection against additive attacks. The reconstruction of the watermark code is only possible when all bits derived from the secret integer code match precisely. As demonstrated in our experiments, altering even a single bit results in a chaotic reconstruction, producing a watermark that no longer corresponds to the original. Consequently, any attempt to embed an additional watermark is treated as tampering and is reliably detected.

6 Added value of watermarking

Traditional integrity verification mechanisms, such as HMAC, play a fundamental role in ensuring that data remains authentic and unaltered during transmission or storage. These techniques operate by generating a cryptographic digest of a message combined with a secret key, enabling recipients to verify both the source and integrity of the data. However, while highly effective at this level, their protective scope is generally limited to external verification and does not extend into the internal execution behavior of a software system.

The watermarking-based approach presented in this work introduces an additional layer of protection that complements, rather than replaces, conventional integrity checks. Its distinctive advantage lies in the way watermark reconstruction is semantically bound to the program's execution. Rather than verifying data as a static entity, the watermark is reconstructed dynamically based on runtime conditions, input parameter relationships, and control-flow behavior. This tight coupling ensures that even subtle modifications, such as changes to internal variables, manipulation of control structures, or tampering with parameter values, disrupt the reconstruction process and reveal the presence of unauthorized alterations.

Another important benefit of watermarking is its capacity for fine-grained detection. Whereas cryptographic mechanisms typically produce a binary outcome indicating whether data is authentic, watermark-based detection provides insight into how and where the system has been tampered with. Because the watermark is embedded across multiple execution points and depends on consistent computational behavior, even small deviations lead to incorrect reconstruction, signaling potential compromise.

Moreover, watermarking offers resilience against forms of manipulation that conventional mechanisms may fail to detect. Operations such as semantic-preserving code transformations, aggressive compiler optimizations, or runtime reordering of computations do not generally affect cryptographic verification results. Yet, because these transformations often disrupt the conditions required for watermark reconstruction, they are readily detected by the proposed approach. This capability extends integrity protection deeper into the software's execution layer, addressing threats that occur beyond the boundaries of data transmission and storage.

In this way, watermarking provides a complementary defense mechanism that strengthens overall system security. Used alongside traditional cryptographic methods, it establishes a multi-layered protection strategy: one layer ensuring the authenticity of data at the communication level, and another safeguarding the integrity of the computation itself. This combination significantly increases resilience against sophisticated tampering attempts and enhances the trustworthiness of the software environment.

7 Conclusions

In this work, a watermarking-based solution was proposed for checking the authenticity of input parameters in high-level programming languages. The approach is built on an extension of the traditional architecture of data structure dynamic software watermarking techniques. Its validation was carried out in Python, demonstrating that the method can detect violations of input authenticity without requiring significant modifications to the protected software or impacting its performance. Future extensions of the proposed architecture could further increase robustness and security by combining dynamic and static watermarking principles.

The applicability of the proposal presented in this work extends beyond verifying the authenticity of input parameters. It can also enhance security in other contexts by checking the validity of exchanged values. For example, it can be used to verify software licenses, ensuring that only the correct keys can assemble the proper watermark and allow the software's use. Additionally, it can support malware analysis by constructing execution graphs in real-world environments outside of sandboxes. Finally, watermarking based on hardware IDs, environments, or configurations further expands the options for tamper detection. Overall, this approach enables indirect and stealthy verification of input authenticity.

As future work, we plan to enhance the stealthiness and blindness of the proposed approach by addressing issues highlighted in this paper, such as integrating static and dynamic techniques and embedding critical parameters within the protected code.

Data availability statement

The original contributions presented in the study are included in the article/supplementary material, further inquiries can be directed to the corresponding author.

Author contributions

MP: Writing – original draft, Writing – review & editing, Conceptualization, Methodology, Software, Validation.

Funding

The author(s) declare that financial support was received for the research and/or publication of this article. This study was partially funded by the European Union - NextGenerationEU, in the framework of the iNEST - Interconnected Nord-Est Innovation Ecosystem (iNEST ECS_00000043 - CUP H43C22000540006), SERICS (PE00000014 - CUP H73C2200089001), and PADS4Health (PRIN PNRR 2022 P2022MSMAW - CUP N. H53D23010880001).

Conflict of interest

The author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Generative AI statement

The author(s) declare that no Gen AI was used in the creation of this manuscript.

Any alternative text (alt text) provided alongside figures in this article has been generated by Frontiers with the support of artificial intelligence and reasonable efforts have been made to ensure accuracy, including review by the authors wherever possible. If you identify any issues, please contact us.

Publisher's note

Author disclaimer

The views and opinions expressed are solely those of the authors and do not necessarily reflect those of the European Union, nor can the European Union be held responsible for them.

References

Alitavoli, M., Joafshani, M., and Erfanian, A. (2013). "A novel watermarking method for java programs," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing* (New York: ACM), 1013–1018.

Alrehily, A., and Thayananthan, V. (2018). Computer security and software watermarking based on return-oriented programming. *Int. J. Comput. Netw. Inf. Secur.* 10, 28–36. doi: 10.5815/ijcnis.2018.05.04

Balduzzi, M., Gimenez, C. T., Balzarotti, D., and Kirda, E. (2011). "Automated discovery of parameter pollution vulnerabilities in web applications," in *Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, USA: NDSS).

Barán, B., Gómez, S., and Bogarin, V. (2001). "Steganographic watermarking for documents," in *Proceedings of the 34th Annual Hawaii International Conference on System Sciences* (Maui, HI: IEEE).

Barker, E., and Kelsey, J. (2012). "Recommendation for random number generation using deterministic random bit generators," in *Technical Report* (Gaithersburg, MD: NIST).

Barni, M., Bartolini, F., and Piva, A. (2001). Improved wavelet-based watermarking through pixel-wise masking. *IEEE Trans. Image Proc.* 10, 783–791. doi: 10.1109/83.918570

Bellare, M., Canetti, R., and Krawczyk, H. (1996). "Keying hash functions for message authentication," in *Advances in Cryptology – CRYPTO '96* (Cham: Springer), 1–15.

Bender, W., Gruhl, D., Morimoto, N., and Lu, A. (1996). Techniques for data hiding. *IBM Syst. J.* 35, 313–336. doi: 10.1147/sj.353.0313

Bento, L. M., Boccardo, D. R., Machado, R. C., Pereira de Sá, V. G., and Szwarcfiter, J. L. (2019). Full characterization of a class of graphs tailored for software watermarking. *Algorithmica* 81, 2899–2916. doi: 10.1007/s00453-019-00557-w

Chan, P. P., Hui, L. C., and Yiu, S.-M. (2012). Heap graph based software theft detection. *IEEE Trans. Inform. Forens. Security* 8, 101–110. doi: 10.1109/TIFS.2012.2223685

Chen, J., Dai, S., and Chen, J. (2016). "An improved software watermarking scheme based on ppct encoding," in *2016 9th International Symposium on Computational Intelligence and Design (ISCID)* (Hangzhou: IEEE).

Chen, Z., Jia, C., and Xu, D. (2017). "Hidden path: dynamic software watermarking based on control flow obfuscation," in *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)* (Guangzhou: IEEE), 443–450.

Chen, Z., Wang, Z., and Jia, C. (2018). Semantic-integrated software watermarking with tamper-proofing. *Multimed. Tools Appl.* 77, 11159–11178. doi: 10.1007/s11042-017-5373-7

Chionis, I., Chroni, M., and Nikolopoulos, S. D. (2014). Waterrpg: a graph-based dynamic watermarking model for software protection. *arXiv* [preprint] arXiv:1403.6658. doi: 10.48550/arXiv.1403.6658

Chiru, C. (2005). "Dynamic software watermarking by altering the numeric results of the program," in *Proceedings of the 11th International Conference on System Theory and Control (SINTES 11)* (Craiova: Spiru Haret University).

Chroni, M., and Nikolopoulos, S. D. (2011). Efficient encoding of watermark numbers as reducible permutation graphs. *arXiv* [preprint] arXiv:1110.1194. doi: 10.1145/2383276.2383295

Collberg, C., Carter, E., Debray, S., Huntwork, A., Kececioglu, J., Linn, C., et al. (2004a). "Dynamic path-based software watermarking," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (New York: ACM), 107–118.

Collberg, C., Jha, S., Tomko, D., and Wang, H. (2004b). *UWStego: A General Architecture for Software Watermarking.*

Collberg, C., Myles, G., and Huntwork, A. (2003). Sandmark-a tool for software protection research. *IEEE Security Privacy* 1, 40–49. doi: 10.1109/MSECP.2003.1219058

Collberg, C., and Thomborson, C. (1998). "On the limits of software watermarking," in *Technical Report* (Auckland: Department of Computer Science, The University of Auckland).

Collberg, C., and Thomborson, C. (1999). "Software watermarking: Models and dynamic embeddings," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 311–324.

Collberg, C. S., Thomborson, C., and Townsend, G. M. (2007). Dynamic graph-based software fingerprinting. *ACM Trans. Program. Lang. Syst.* 29:35-es. doi: 10.1145/1286821.1286826

Cox, I., Miller, M., Bloom, J., Fridrich, J., and Kalker, T. (2007). *Digital Watermarking and Steganography*. San Francisco, CA: Morgan Kaufmann Publishers Inc.

Cox, I. J., Miller, M. L., Bloom, J. A., Fridrich, J., and Kalker, T. (2008). *Digital Watermarking*. Burlington, MA: Morgan Kaufmann Publishers, 56–59.

Curran, D., Cinneide, M., Hurley, N., and Silvestre, G. (2004). "Dependency in software watermarking," in *Proceedings. 2004 International Conference on Information and Communication Technologies: From Theory to Applications* (Damascus: IEEE), 569–570.

Dalla Preda, M., Giacobazzi, R., and Visentini, E. (2008). "Hiding software watermarks in loop structures," in *Static Analysis: 15th International Symposium, SAS 2008* (Valencia, Spain: Springer), 174–188.

Dalla Preda, M., and Ianni, M. (2024). Exploiting number theory for dynamic software watermarking. *J. Comp. Virol. Hack. Tech.* 20, 41–51. doi: 10.1007/s11416-023-00489-8

Dalla Preda, M., and Pasqua, M. (2019). Semantics-based software watermarking by abstract interpretation. *Mathem. Struct. Comp. Sci.* 29, 339–388. doi: 10.1017/S0960129518000038

Davidson, R. I., and Myhrvold, N. (1996). *Method and System for Generating and Auditing a Signature for a Computer Program.*

Desmedt, Y. (2025). "Secret sharing and shamir threshold scheme," in *Encyclopedia of Cryptography, Security and Privacy* (Cham: Springer), 2212–2216.

Dey, A., Bhattacharya, S., and Chaki, N. (2019). Software watermarking: progress and challenges. *INAE Letters* 4, 65–75. doi: 10.1007/s41403-018-0058-8

Dey, A., Ghosh, S., Bhattacharya, S., and Chaki, N. (2020). A robust software watermarking framework using shellcode. *Multimed. Tools Appl.* 79, 2555–2576. doi: 10.1007/s11042-019-08372-9

Durstenfeld, R. (1964). Algorithm 235: Random permutation. *Commun. ACM* 7:420. doi: 10.1145/364520.364540

Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognit. Lett.* 27, 861–874. doi: 10.1016/j.patrec.2005.10.010

Gupta, G., and Pieprzyk, J. (2006). "A low-cost attack on branch-based software watermarking schemes," in *Digital Watermarking: 5th International Workshop, IWDW 2006* (Jeju Island, Korea: Springer), 282–293.

Gupta, G., and Pieprzyk, J. (2007). Software watermarking resilient to debugging attacks. *J. Multimed.* 2, 10–16. doi: 10.4304/jmm.2.2.10-16

Gupta, G., and Pieprzyk, J. (2009). Reversible and blind database watermarking using difference expansion. *Int. J. Digital Crime Forensics (IJDCF)* 1, 42–54. doi: 10.4018/jdcf.2009040104

Halder, R., Pal, S., and Cortesi, A. (2010). Watermarking techniques for relational databases: Survey, classification and comparison. *J. Univer. Comp. Sci.* 16, 3164–3190.

Hamilton, J., and Danicic, S. (2010). "An evaluation of static java bytecode watermarking," in *Proceedings of the International Conference on Computer Science and Applications (ICCSA'10)*, *The World Congress on Engineering and Computer Science (WCECS'10)* (San Francisco: ICCSA).

Katz, J., Menezes, A. J., Van Oorschot, P. C., and Vanstone, S. A. (1996). *Handbook of Applied Cryptography*. Boca Raton, FL: CRC Press.

Katzenbeisser, S., and Petitcolas, F. (2016). *Information Hiding.* Norwood: Artech House.

Ke-xin, Y., Ke, Y., and Jian-qi, Z. (2009). "A robust dynamic software watermarking," in *2009 International Conference on Information Technology and Computer Science* (Kiev: IEEE), 15–18.

Kim, T., Jang, Y., Lee, C., Koo, H., and Kim, H. (2023). "SmartMark: software watermarking scheme for smart contracts," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (Melbourne: IEEE), 283–294.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Boston, MA: Addison-Wesley.

Krawczyk, H., Bellare, M., and Canetti, R. (1997). "Hmac: Keyed-hashing for message authentication," in *RFC 2104, Internet Engineering Task Force*.

Krawczyk, H., and Eronen, P. (2010). "HMAC-based extract-and-expand key derivation function (HKDF)," in *RFC 5869, IETF*.

Kumar, K., Kehar, V., and Kaur, P. (2015). A comparative analysis of static java bytecode software watermarking algorithms. *Afric. J Comput ICT* 8, 201–208.

Lee, T., Hong, S., Ahn, J., Hong, I., Lee, H., Yun, S., et al. (2023). Who wrote this code? watermarking for code generation. *arXiv* [preprint] arXiv:2305.15060. doi: 10.18653/v1/2024.acl-long.268

Lemire, D. (2019). Fast random integer generation in an interval. ACM Trans. Model. Comp. Simulat. (TOMACS) 29, 3:1–3:12. doi: 10.1145/3230636

Ma, H., Jia, C., Li, S., Zheng, W., and Wu, D. (2019). Xmark: dynamic software watermarking using collatz conjecture. *IEEE Trans. Inform. Forens. Secur.* 14, 2859–2874. doi: 10.1109/TIFS.2019.2908071

Ma, H., Lu, K., Ma, X., Zhang, H., Jia, C., and Gao, D. (2015). "Software watermarking using return-oriented programming," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 369–380.

Madou, M., Anckaert, B., De Sutter, B., and De Bosschere, K. (2005). "Hybrid static-dynamic attacks against software protection mechanisms," in *Proceedings of the 5th ACM Workshop on Digital Rights Management*, 75–82.

Myles, G., and Collberg, C. (2006). Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Res.* 6, 155–171. doi: 10.1007/s10660-006-6955-z

Myles, G., and Jin, H. (2005). "Self-validating branch-based software watermarking," in *International Workshop on Information Hiding* (Cham: Springer), 342–356.

Palsberg, J., Krishnaswamy, S., Kwon, M., Ma, D., Shao, Q., and Zhang, Y. (2000). "Experience with software watermarking," in *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)* (New Orleans: IEEE), 308–316.

Patel, S. J., and Pattewar, T. M. (2014). "Software birthmark based theft detection of javascript programs using agglomerative clustering and frequent subgraph mining," in *2014 International Conference on Embedded Systems (ICES)* (Coimbatore: IEEE), 63–68.

Pérez Gort, M. L., Feregrino Uribe, C., and Nummenmaa, J. (2017). "A minimum distortion: High capacity watermarking technique for relational data," in *Proceedings of the 5th ACM Workshop on Information Hiding and Multimedia Security*, 111–121.

Pérez Gort, M. L., Feregrino-Uribe, C., Cortesi, A., and Fernandez-Pena, F. (2020). A double fragmentation approach for improving virtual primary key-based watermark synchronization. *IEEE Access* 8, 61504–61516. doi: 10.1109/ACCESS.2020.2979659

Pérez Gort, M. L., Olliaro, M., and Cortesi, A. (2021). "A quantile-based watermarking approach for distortion minimization," in *International Symposium on Foundations and Practice of Security* (Cham: Springer), 162–176.

Pieprzyk, J. (1999). "Fingerprints for copyright software protection," in *Information Security: Second International Workshop, ISW'99* (Kuala Lumpur: Springer), 178–190.

Pornin, T. (2013). "Deterministic usage of the digital signature algorithm (DSA) and elliptic curve digital signature algorithm (ECDSA)," in *RFC 6979, IETF*.

Provos, N., and Honeyman, P. (2003). Hide and seek: an introduction to steganography. *IEEE Security Privacy* 1, 32–44. doi: 10.1109/MSECP.2003.1203220

Sion, R., Atallah, M., and Prabhakar, S. (2003). "Rights protection for relational data," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 98–109.

Sokolova, M., and Lapalme, G. (2009). A systematic analysis of performance measures for classification tasks. *Inform. Proc. Managem.* 45, 427–437. doi: 10.1016/j.ipm.2009.03.002

Stern, J. P., Hachez, G., Koeune, F., and Quisquater, J.-J. (2000). "Robust object watermarking: application to code," in *Information Hiding: Third International Workshop, IH'99* (Dresden: Springer), 368–378.

Tamada, H., Okamoto, K., Nakamura, M., Monden, A., and Matsumoto, K. (2004). "Dynamic software birthmarks to detect the theft of windows applications," in *International Symposium on Future Software Technology, volume 20*.

Thomborson, C., Nagra, J., Somaraju, R., and He, C. (2004). "Tamper-proofing software watermarks," in *Proceedings of the Second Workshop on Australasian Information Security, Data Mining and Web Intelligence, and Software Internationalisation-Volume 32*, 27–36.

Tian, J. (2003). Reversible data embedding using a difference expansion. *IEEE Trans. Circuits Syst. Video Technol.* 13, 890–896. doi: 10.1109/TCSVT.2003.815962

Tian, Z., Zheng, Q., Liu, T., Fan, M., Zhuang, E., and Yang, Z. (2015). Software plagiarism detection with birthmarks based on dynamic key instruction sequences. *IEEE Trans. Softw. Eng.* 41, 1217–1235. doi: 10.1109/TSE.2015.2454508

Vivekananthan, V., Praveen, K., and Sethumadhavan, M. (2021). "Dynamic watermarking using python ast," in *Proceedings of International Conference on Advances in Computer Engineering and Communication Systems: ICACECS 2020* (Cham: Springer), 219–231.

Wang, Y., Gong, D., Lu, B., Xiang, F., and Liu, F. (2018). Exception handling-based dynamic software watermarking. *IEEE Access* 6, 8882–8889. doi: 10.1109/ACCESS.2018.2810058

Wu, B., Chen, K., He, Y., Chen, G., Zhang, W., and Yu, N. (2024). "Codewmbench: An automated benchmark for code watermarking evaluation," in *Proceedings of the ACM Turing Award Celebration Conference-China 2024*, 120–125.