Xmark: Dynamic Software Watermarking Using Collatz Conjecture

Haoyu Ma, Chunfu Jia, Shijia Li, Wantong Zheng, and Dinghao Wu

***Abstract*— Dynamic software watermarking is one of the major countermeasures against software licensing violations. However, conventional dynamic watermarking approaches have exhibited a number of weaknesses including exploitable payload semantics, exploitable embedding/recognition procedures, and weak correlation between payload and subject software. This paper presents a novel dynamic watermarking method, `Xmark`, which leverages a well-known unsolved mathematical problem referred to as the Collatz conjecture. Our method works by transforming selected conditional constructs (which originally belonged to the software to be watermarked) with a control flow obfuscation technique based on Collatz conjecture. These obfuscation routines are built in a particular way such that they are able to express a watermark in the form of iteratively executed branching activities occurred during computing the aforementioned conjecture. Exploiting the one-to-one correspondence between natural numbers and their orbits computed by the conjecture (also known as the "Hailstone sequences"), `Xmark`'s watermark-related activities are designed to be insignificant without the pre-defined secret input. Meanwhile, being integrated with obfuscation techniques, our method is able to resist attacks based on various reverse engineering techniques on both syntax and semantic levels. Analyses and simulations indicated that `Xmark` could evade detections via pattern matching and model checking, and meanwhile effectively prohibit dynamic symbolic execution. We have also shown that our method could remain robust even if a watermarked software is compromised via re-obfuscation using approaches like control flow flattening.**

***Index Terms*— Software watermarking, code obfuscation, Collatz conjecture.**

I. Introduction

FOR a long time, software industry has been at war with intellectual property violation of various kinds, including software piracy, plagiarism and etc. A recent report from The Software Alliance said that as of 2017, there are still 37% of all software installed worldwide which are not properly licensed, and the commercial value loss due to the unlicensed software can be as high as $46.3 billion [1]. Meanwhile, the rising of mobile platforms in recent years has greatly encouraged app piracy, which have started siphoning huge amount of revenue from legitimate mobile publishers. The need of maintaining a health ecosystem in software market calls for countermeasures against software piracy and others. Inheriting the idea of digital watermarking, people have brought out the so called *software watermarking* technique to meet the request.

A. Prior Work

Many software watermarking schemes have been proposed in the last several decades [2]–[4]. Formally, a software watermarking system is a collection of functions $\mathfrak{W}:\{E(\cdot), R(\cdot)\}$, consisting of a embedding process $E(\cdot)$ (a program transformation in essence) and a recognition protocol $R(\cdot)$. Given a subject software $\mathbf{S}$, a watermark w and a secret i as w's trigger, $\mathfrak{W}$ produces the watermarked software instance $\mathbf{S}_w$ by

$$E(\mathbf{S}) \xrightarrow{w,i} \mathbf{S}_w. \tag{1}$$

Later, using the same i, the recognition protocol of $\mathfrak{W}$ should be able to reliably output w from $\mathbf{S}_w$ by computing

$$R(\mathbf{S}_w) \xrightarrow{i} w. \tag{2}$$

There is also a widely accepted set of functional and security goals for software watermarking designs, namely:

- *stealth*, an embedded watermark message should exist as an inconspicuous element of the subject software;
- *resilience*, an embedded watermark should remain functional even if the subject software suffers from a various kinds of manipulations intended to destroy it;
- *data rate*, which describes the cost (in term of code bloat) for embedding one bit of watermark; and
- *performance*, which describes the overhead caused to the subject software after a watermark is embedded.

A software watermarking method can be static or dynamic. *Static watermarking* hides watermarks in code and/or data of a software [5]–[12], yet methods of this type are known to be highly susceptible to attacks based on semantic-preserving transformations [13]. In *dynamic watermarking*, a watermark is instead turned into data objects or program states created by executing well-crafted *payload code*. The payload is then injected into the subject software to be activated only when it is run using a pre-selected special input-lineup (known as the *secret input*) [14]–[25]. Dynamic watermarking is considered a better option since it is by nature able to ignore many generic semantic-preserving transformations. However, by reviewing details of existing dynamic watermarking methods and some targeted attack schemes, we find many known designs of this type flawed at least on some aspects of their effectiveness.

Manuscript received August 2, 2018; revised January 27, 2019 and March 24, 2019; accepted March 26, 2019. Date of publication March 29, 2019; date of current version June 27, 2019. This work was supported in part by the National Natural Science Foundation of China under Grant 61702399 and Grant 61772291 and in part by the Natural Science Foundation of Tianjin, China, under Grant 17JCZDJC30500. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Tomas Pevny. *(Corresponding author: Chunfu Jia.)*

H. Ma is with Xidian University, Xi'an 710126, China (e-mail: hyma@xidian.edu.cn).

C. Jia, S. Li and W. Zheng are with Nankai University, Tianjin 300071, China (e-mail: cfjia@nankai.edu.cn; sjli@mail.nankai.edu.cn; zhengwantong@mail.nankai.edu.cn).

D. Wu is with Pennsylvania State University, University Park, PA 16802 USA (e-mail: dwu@ist.psu.edu).

Digital Object Identifier 10.1109/TIFS.2019.2908071

The first category of design weaknesses is that watermark payloads constructed by many existing works possess certain *exploitable semantics* as intrinsic elements, which significantly undermines their stealth. Typical examples include the path- and thread-based watermarking techniques [15], [16], [20]. These methods encoded watermarks bit-by-bit with two types of basic code widgets representing `0`s and `1`s. The resulting payloads therefore repeated their widgets in unusual patterns, leaving distinct footprints for malicious recognition attempts. It was pointed out that the density of branches or function calls introduced by path-based watermarking is too abnormal for these code pieces to be taken for natural program logic [3]. In fact, same observation was already adopted as the basis of a targeted (and successful) attack [26]. Similarly, to improve stealth, thread-based watermarking created deceptive contexts with a large number of decoy widgets because basic widgets in constructed are too noticeable on themselves [16]. Tradeoffs of *using decoys*, however, were significant overhead and code bloat which limited the applicability of this method [3]. Other cases of this category involve methods that caused noticeable abnormalities in order to deploy their payloads. For instance, a branch-based watermarking introduced fingerprint branch functions (FBFs) into the subject software as both control flow obfuscation and watermark constructor [17]. However, doing so changes structure of targeted functions in a suspicious way, causing the FBFs be exposed to further attacks [27].

Secondly, as pointed out in [25], payload constructed by many existing dynamic watermarking methods tend to have *weak correlation* with softwares they are merged with. Such as, in graph-based watermarking and its derivatives [14], [19], [22], payload widgets were designed to have nothing else to do with the subject software except being attached to it. Having weak correlation is not directly exploitable. Instead, it impairs a dynamic watermark by making its security overly reliance on stealth — because once located, payload corresponding to such a watermark can be safely pruned with negligible collateral damage, making resilience against generic attacks meaningless under the circumstances. Existing attacks targeting path- and branch-based watermarking had showed that defeating designs with defective stealth can be much easier than expected if they failed to establish proper dependencies between their payloads and subject softwares [26], [27]. A similar case is the opaque predicate based watermarking (and its static origin) [7], [18]. These designs used invariant opaque predicates as carriers of watermark segments, with the assumption that distinguishing such predicates apart from meaningful control flow structures is difficult. Unfortunately, later studies soon provided effective and efficient means to detect invariant opaque predicates [28], [29], at which point the use of opaque predicates became a major shortcoming because being tautologies in nature makes the removing of these routines both safe and logical. Another evidence is a more recent non-stealthy dynamic watermarking method for Android apps [23], which relied entirely on code hiding and obfuscation to protect its payload. The observation of this work indicated that a so-concealed watermark widget immediately becomes defenseless after it is eventually released at runtime. Determined adversaries can therefore resort to a combination of various attacks to remove the payload, even though this method has adopted *redundant embedding* strategy to enhance survivability of its watermark.

Last but not least, the third category of weaknesses exist in several previous designs is the use of *exploitable procedures*. In other words, certain mechanisms defined in the embedding and/or recognition protocols of these methods could be taken advantage of to launch tailor-made attacks on them. The first case we identified is the hash function based watermarking method [21]. The recognition protocol of this method involved tracing the subject software with a debugger and monitoring functions with 4 parameters. It was also defined to overwrite the last parameter of matched functions (with 0), and fetch their return values as potential watermark pieces. This allows adversaries to identify the same functions of interest, and hook them to a proxy which detects the recognizer with signs of debugging and values of certain parameters as features. The targeted recognition process can then invalidated by corrupting return values of the hooked functions. Monitoring/modifying a program's execution using function and API hooking is well studied [30], which strongly indicates the feasibility of this attack. Similarly, a more recent work which adopted return-oriented programming (ROP) for watermarking also exposed weakness of this type [24]. The recognition procedure of this method was designed to be triggered by overwriting a function pointer to direct control to ROP payloads. Thus adversaries can expose such payloads by hooking all indirect function calls of a software to analyze code of the callees for ROP behaviors.

B. The `Xmark` Scheme

This paper presents a novel dynamic software watermarking scheme, named `Xmark`, which is established on top of a code obfuscation technique based on unsolved conjectures [31]. Our method performs a specialized obfuscation on a collection of conditional constructs owned by the subject software to build its payload. The watermark is parsed into a sequence of natural numbers, which are used to control the obfuscation routines of our method when the watermarked software runs with specific input cases defined in the combination of secret input. We let behavioral patterns of such payload routines be intentionally echoed across multiple execution traces driven by the secret input, making these patterns recognizable as the consequence.

The overall objective we intend to achieve with the `Xmark` scheme is a solution that not only satisfies the generic requirements for dynamic watermarking, but also avoids the particular weaknesses as described in Section I-A. Specifically:

- payload of our method should be difficult to locate using either static or dynamic analyses;
- our method should be able to resist attempts of removing its payload via de-obfuscation based on state-of-the-art reverse engineering techniques;
- our recognition protocol should remain robust even if the payload constructs suffer from malicious re-obfuscation regarding to properties like control structure;
- a basic unit of our payload constructs should be able to carry a watermark that is longer than a machine integer;
- performance overhead and code bloat due to embedding a watermark using our method should be acceptable.

TABLE I
A COMPARISON ON TYPICAL DYNAMIC WATERMARKING SCHEMES (REGARDING WEAKNESSES)

	exploitable semantics	exploitable procedures	weak correlation	using decoys	redundant embedding
Graph based watermark [21]			×		
Path based watermark [17]	×		×		×
Branch based watermark [19]	×		×		
Opaque predicate watermark [20]			×		
Threading watermark [18]	×			×	
Hash based watermark [23]		×			
Droidmarking [25]	×		×		×
ROP based watermark [26]		×	×		
Neuroprint [27]				×	
Xmark					

`Xmark` does not require locating its payload as precondition of a successful watermark recognition, thus its recognizer is not designed to look for some particular static code features that might be exploited by adversaries. Its recognition protocol also determines that the embedded watermark pieces are not given away just by hitting some of the secret-input-driven execution paths by luck. As obfuscated predicates, our payload objects cannot be straight-forwardly removed since their existence is necessary with regard to the subject software's integrity. The underlying obfuscation technique of `Xmark` can be identified as a special type of dynamic opaque predicate [29]. It exploits an intrinsic shortcoming of symbolic execution, causing state explosion to prohibit dynamic program analyses. As the result, resilience of `Xmark` against state-of-the-art reverse engineering tools is significantly improved, especially when compared with similar schemes based on invariant opaque predicates [7], [18]. To the best of our knowledge, `Xmark` is the first of its kind to overcome all types of weaknesses stated in Section I-A, as summarized in Table I.

II. BACKGROUND

A. Control Flow Obfuscation Using Unsolved Conjectures

Unsolved conjectures are important elements in the history of mathematics, and *Collatz conjecture*, or the *3x+1 problem*, is no doubt a representative example. The core of this problem is the so-called *Collatz function*, a mapping $\theta : \mathbf{N}^* \to \mathbf{N}^*$ where for any $n \in \mathbf{N}^*$,

$$\theta(n) = \begin{cases} n \div 2 & if\ n\ is\ even, \\ 3n+1 & if\ n\ is\ odd. \end{cases} \tag{3}$$

Let $\theta^0(n) = n$, and let

$$\theta^k(n) = \underbrace{\theta \circ \cdots \circ \theta(n)}_{k \text{ times}} = \theta(\theta^{k-1}(n)), \tag{4}$$

Collatz conjecture asserts that there always exists a $\delta_n \in \mathbf{N}^*$ such that $\theta^{\delta_n}(n) = 1$. Note that the iterative computing given in Equation 4 derives a sequence $\Lambda(n) = \{\theta^k(n)\}_{k=0}^{\delta_n}$, which is called the *orbit* for n in the conjecture (also known as n's *Hailstone sequence*). The assertion of Collatz conjecture also implies that, given $n_1, n_2 \in \mathbf{N}^*$, $\Lambda(n_1) = \Lambda(n_2)$ holds only if $n_1 = n_2$, i.e. the number of distinct Hailstone sequences is as many as that of members in $\mathbf{N}^*$.

Collatz conjecture is "a deterministic process that simulates 'random' behavior" [32]. This characteristic was thus adopted in a control flow obfuscation method [31]. Figure 1 illustrates the main idea of the obfuscation. Let the subject be a branch which directs control to a `Do_sth()` module if its predicate "$x == C$" evaluates to true (as in Figure 1.a). The obfuscation first creates a spurious integer variable $y > 0$ via a mapping denoted by $\phi(x)$ (with the original condition variable x as the seed), then attach a Collatz conjecture loop controlled by y to participate predicate evaluations of the branch, resulting in a routine as illustrated in Figure 1.b. This transformation asserts correctness of the subject branch by replacing its predicate (as aforementioned) with a combinatory logic $eval(x, y, C)$ which instead verifies whether

$$\begin{cases} x + y < C + 2 \\ x - y > C - 2 \end{cases} \tag{5}$$

is satisfied. The obfuscated routine operates iteratively, in each round it computes $y = \theta(y)$ before evaluating $eval(x, y, C)$. Equation 5 indicates that $eval(x, y, C)$ is not satisfiable until y yields to 1, at which point the evaluated predicate becomes "$C - 1 < x < C + 1$", which (for integers) is equivalent to the original predicate. However, since y is not a concrete value, adversaries trying to explore the protected branch has to resort to symbolic execution where y is handled as a symbol. As the result, the loop of Collatz conjecture introduces numerous of potential execution paths, crippling the analysis before it solves the actual branch bounder. In this paper, we use the above obfuscation method as the corner stone of `Xmark`'s watermark payload constructs.

B. Mixed Boolean-Arithmetic Encoding

Another code obfuscation technique involved in this work is the *mixed boolean-arithmetic (MBA) encoding*, which was proposed to provide diversified obscuring transformations on real-world functions and data via mixed-mode computation over Boolean-arithmetic algebras [33]. By constructing the so-called *zero/invertible MBA functions*, MBA encoding can be used to hide constants or variables, and consequently conceal actual semantics of algorithms. A main resource for building such MBA functions is the *permutation polynomials* [34]. According to Theorem 3 of [33], a polynomial over $Z/(2^n)$ is a permutation polynomial (and therefore invertible) if its 1st-order coefficient is odd and all higher order coefficients are even. An example provided in [33] is the inverse generation for cubic polynomials: given a $f(x) = \sum_{i=0}^{3} a_i x^i$, its inverse

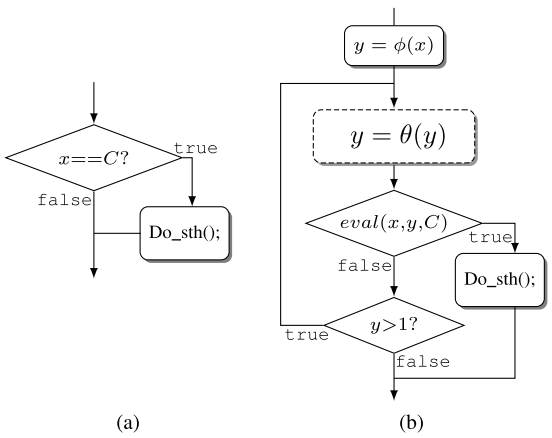


Fig. 1. Basic idea of the Collatz-conjecture-based control flow obfuscation. (a) Original structure. (b) Obfuscated version.

$f^{-1}(x) = \sum_{i=0}^{3} a'_i x^i$ can be computed by

$$\begin{cases} a'_3 = -a_1^{-4} a_3 \\ a'_2 = -a_2 a_1^{-3} + 3a_0 a_1^{-4} a_3 \\ a'_1 = a_1^{-1} + 2a_0 a_1^{-3} a_2 - 3a_0^2 a_1^{-4} a_3 \\ a'_0 = -a_0 a_1^{-1} - a_0^2 a_1^{-3} a_2 + a_0^3 a_1^{-4} a_3. \end{cases} \tag{6}$$

We exploit these existing results to support the construction of Xmark's watermark payload constructs.

C. Adversarial Model Against Xmark

For the soundness of this work, we define the adversarial model against Xmark on both the *priori knowledge* and the *strategies* of potential adversaries.

1) Priori Knowledge: An implication made by many past researches (via the assumed attacks against their designs [15], [16], [19], [23]–[25]) is that *adversaries against a software watermarking scheme should be considered as being aware of all deterministic design details regarding both its embedding and recognition processes*. In this paper, we too consider the same assumption since it is practically reasonable. Nevertheless, the capacity of adversaries we assumed are also limited on a number of aspects. To start with, *adversaries should have no knowledge on the secrets used by the watermarking scheme beforehand* (e.g. the secret i defined in Equation 1 and 2 from Section I-A). In specific, two types of secrets are referred to in this assumption:

- any secret used by the target scheme in the extraction of an embedded watermark; and
- any secret that is involved in the process of embedding a watermark with the target scheme.

To give an example on the second type of secrets, consider a watermark embedder which randomizes its payload in some way. The so introduced randomness should be a secret to the assumed adversaries, i.e. although they know the fact that the targeted payload is randomized as well as the method used to do so, concrete implementation of this randomized payload is still something out of their reach. Moreover, like suggested in [19], given a subject software, we assume that *adversaries have not determined in advance on whether the subject indeed carries a watermark*, i.e. a diagnose regarding the existence of watermark is needed before they can take any further moves. Last but not least, *adversaries are assumed to have no access to source code of subject softwares (either the original or watermarked version)*. Here we assume that subject softwares are released as commercial off-the-shelf (COTS) binaries, thus the adversary can only launch attacks on binary level.

2) Strategies: As suggested by existing works [14]–[16], [19], the generic threat model against dynamic watermarking consists of *watermark cropping*, *watermark forgery* or *watermark distorting* attacks. Again, use the notations in Equation 1 and 2 to help explaining the concept of these attacks:

- a watermark cropping attack aims to transform $\mathbf{S}_w$ in a way that elements related to w are removed while useful semantics of $\mathbf{S}$ are preserved;
- a watermark forgery attack aims to deceive the recognizer $R(\cdot)$ into believing that a compromised instance of $\mathbf{S}_w$ carries a bogus watermark $w' \neq w$; and
- a watermark distorting attack, while also attempt to make w unrecognizable, applies indiscriminate transformations on $\mathbf{S}_w$ rather than targeting specific elements of it.

Since watermark forgery could be deemed as successful even if w still survives after $\mathbf{S}_w$ is compromised, we consider coping with such attacks more as a tamper-proofing task and thus out of our scope. Nevertheless, we include *watermark disclosure* attacks into the arsenal of adversaries. The aim of watermark disclosure is to reveal those elements of $\mathbf{S}_w$ that are relevant to w with the absence of secret i, so that subsequent attacks can be properly focused on the revealed elements.[1] Therefore, we consider such attacks as the necessary premise of watermark cropping attacks. Our specific adversarial model focuses on a number of targeted attacks which are by all means adapted version of the considered generic attacks, namely:

- we consider *pattern matching* and *model checking* as the static-level tools of watermark disclosure attacks against Xmark. In particular, model checking should be utilized in the same way as in malicious code detection [37];
- we also consider a dynamic *probing attack* against our method, with adversaries imitating the same protocol of our recognizer while exploring execution traces of the subject software, hoping to detect patterns indicating the existence of Collatz conjecture routines;
- for more powerful adversaries mounted with sophisticated dynamic program analysis tools like *symbolic execution*, we consider watermark cropping attacks against Xmark by removing its payload constructs via de-obfuscation;
- last but not least, being a control-flow based scheme, we also have to consider distortive attacks which attempt to invalidate Xmark by re-transforming the watermarked software via tools like *control flow flattening* [38].

Note that as long as a determined adversary is willing to throw in time and effort, a software-based protection technique (like Xmark) could always be dismantled eventually. Therefore,

[1] In the prior literature, disclosure attack (such as key/memory/statistical disclosure) was used to refer to an adversary's gain in knowledge about cryptographic secrets [35], [36]. Here we extend the concept to cover steganographic secrets, e.g. whether a watermark is present in a covertext (in our scenario, a software), and if so, where the watermark can be found.

we deem our method as effective should the cost for defeating it be great enough to encourage adversaries to consider developing the protected software functionalities from scratch.

III. Method Overview

Recall that the goal of this work is to present a novel design of dynamic software watermarking that is 1) difficult to be disclosed by means like pattern matching and model checking, 2) secure against de-obfuscation attempts using tools such as symbolic execution, 3) robust against certain re-obfuscation approaches, and 4) without obvious drawbacks with respect to data rate and performance (as discussed in Section I-B). The `Xmark` scheme proposed in this paper is established on top of the Collatz-conjecture-based code obfuscation method that is briefly introduced in Section II-A. The key motivation is: structureless and "randomness" (which is exploited by the mentioned obfuscation) is just one aspect of the conjecture, the case-by-case determinacy of its procedure in mapping natural numbers to Hailstone sequences, on the other hand, indicates a potent way to stenographically interpret an arbitrary message into dynamic program activities. We thus aim to fully develop the potential of Collatz conjecture in information hiding.

Figure 2 presents a motivating example for illustrating the idea of our method. Assume a software $\mathbf{S}(\mu)$ where μ denotes its input; and, without loss of generality, assume the goal is to embed a watermark $w=3$ into $\mathbf{S}$ and later recognize it. `Xmark` selects from $\mathbf{S}$ a pair of conditional constructs (i.e. b_1 and b_2 in this example) that can be reached when running $\mathbf{S}$ with input cases $\mu=i_1$ and $\mu=i_2$ respectively. Both constructs are then transformed using a specialized Collatz-conjecture-based obfuscation to produce $\mathbf{S}$'s watermarked instance (denoted by $\mathbf{S}_w$).[2] Different from what's described in Section II-A, here `Xmark` defines a pair of functions $\{\phi_1, \phi_2\}$ which each takes μ and the original condition variable of b_1/b_2 (i.e. x_1 or x_2) as inputs, and outputs spurious variables y_1/y_2 for the Collatz conjecture routines used in obfuscation. These two functions are also constructed to satisfy

$$\phi_1(x_1, i_1) = \phi_2(x_2, i_2) = w. \tag{7}$$

As the result, when running $\mathbf{S}_w(i_1)$, the obfuscated construct labeled as l_1 is reached with $y_1=w=3$. According to Collatz conjecture, y_1 yields to 1 according to the orbit $3\rightarrow 10\rightarrow 5\rightarrow 16\rightarrow 8\rightarrow 4\rightarrow 2\rightarrow 1$. In other words, l_1 would iterate for 7 rounds, with the bold `if-else` branch within the loop takes its `if` path (henceforth the *odd path*) in the 1st and 3rd round, whereas the `else` path (henceforth the *even path*) is taken in the rest rounds. Equation 7 further indicates that *if run $\mathbf{S}_w$ one more time using input i_2, the exact same sequence of branchings will be repeated by the bold* `if-else` *structure inside construct l_2*. `Xmark` identifies the embedded watermark via such intentionally crafted echoing of branching behaviors, then recover the message ($w=3$ in this example) by reversing the corresponding Hailstone sequence. Conditional constructs selected for watermark embedding are henceforth called the *embedding points* for our method. Meanwhile, since the timing of watermark presentation is controlled by ϕ_0 and ϕ_1, they are hereby referred to as the *control units* of `Xmark`'s payload.

[2]For simplicity, in this example we only assume the typical implementation of Collatz conjecture. More extensions regarding to `Xmark`'s embedding transformation are explained in Section III-A.

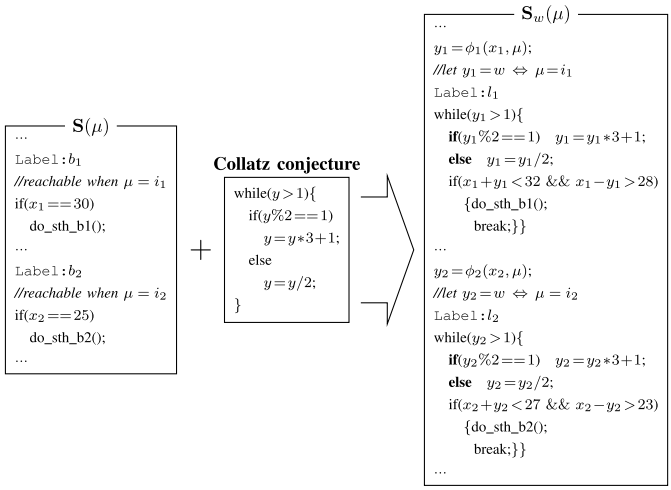


Fig. 2. A motivating example of `Xmark`'s general idea.

Figure 3 gives the overall deployment model of `Xmark`. Given the source of a subject software, our embedder first run a preliminary analysis to determine available embedding points, and subsequently obtain a secret input lineup for reaching them at runtime. These supportive information, together with the watermark message, are then used by a program rewriter to generate binary of the software's watermarked instance. The recognizer of our method, extracts the hidden watermark by running the watermarked binary with the secret input, while tracing and analyzing control transfers within the resulting execution traces. In the rest of this section, we explain the important details regarding to both the watermark embedding process and the recognition protocol of our method.

A. Watermark Embedding

The actual watermark embedding process of `Xmark` is an extension of the aforementioned basic example. Purposes of the extension include making the process compatible with real watermark instances in practice, while preventing the created payload constructs from exposing exploitable features.

1) Embedding Large Watermark: In a real-world scenario, a watermark containing sufficient information is almost always too lengthy to be represented using a single machine integer. Like many existing works, in such a generic scenario, `Xmark` considers w as a collection of substrings $\Omega_w=\{w_\varepsilon|1\leq\varepsilon\leq n\}$, with each w_ε be of a safe length (which is further discussed in Section V-B) to encode into a single unsigned integer. The embedding of w thus yields to that of Ω_w. `Xmark` does this by preparing a long enough input-lineup $\mathcal{I}=i_0\cup\{i_\varepsilon|1\leq\varepsilon\leq n\}$, and extends its control units to further satisfy

$$\phi_1(x_1, i_{\varepsilon-1}) = \phi_2(x_2, i_\varepsilon) = w_\varepsilon, \quad 1\leq\varepsilon\leq n. \tag{8}$$

Extended control units can either be generated using function fitting, or simply implemented as segmented functions. Recall the example in Figure 2, and now assume the goal is instead to embed a longer watermark such as $\Omega_w=\{103, 93\}$ (which

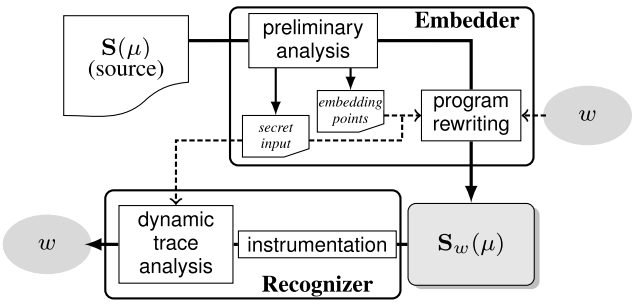


Fig. 3. Deployment model of the Xmark scheme.

is the ASCII of "WM"). Without loss of generality, our method derivates a secret input lineup $\mathcal{I} = \{8, 5, 2018\}$, then

- let ϕ_1 go through coordinates $(8, 103)$ and $(5, 93)$;
- let ϕ_2 go through $(5, 103)$ and $(2018, 93)$;

As the result, after sequentially running $\mathbf{S}_w(8)$, $\mathbf{S}_w(5)$ and $\mathbf{S}_w(2018)$, Xmark should identify $w_1 = 103$ from the traces of $\mathbf{S}_w(8)$ and $\mathbf{S}_w(5)$, then detect $w_2 = 93$ from those of $\mathbf{S}_w(5)$ and $\mathbf{S}_w(2018)$ — both via the same pair of embedding points (i.e. the obfuscated constructs l_1 and l_2). Making watermark substrings "overlap" in such a way improves both stealth and data rate of Xmark, although developers are certainly allowed to embed a watermark into more embedding points.

2) Burying Footprints of Collatz Conjecture: The pivotal operation of Xmark's payload is the Collatz function. In its typical form, this function is of a simple and fixed structure, while at runtime, it generates Hailstone sequences as explicit program states with the loops of Collatz conjecture unrolling. Such fundamental semantic-level features could undermine the stealth of our payload constructs on two aspects:

- structural signatures of the Collatz function could expose the payload constructs to pattern matching; and
- the Hailstone sequences derived by computing the Collatz function could also be picked up using data flow analyses, which consequently exposes our payload as well.

Intuitively, binary mutation and constant/variable obfuscation could be solutions for this problem [39], [40]. However, known techniques of such type have the basic need of being semantic-preserving program transformations [41], whereas in Xmark, features that needs to be concealed exist in semantic-level and therefore can hardly be hidden using such transformations. Our idea of solving the above mentioned problem come from the observation that *for either watermarking or obfuscation, the functionality of* Xmark*'s payload constructs does not rely on the specific operations done in the two paths of their Collatz function routines*:

- the key factor in watermarking is *parity of the spurious variable during the loop unrolling process of a payload construct* (so that its Collatz function branches correctly in every round), whether concrete value of this variable comply to Hailstone sequences is in fact irrelevant;
- regarding obfuscation, a payload construct is able to work correctly as long as both its loop guard and the obfuscated conditional logic it holds behave as expected.

Therefore in Xmark, we apply what we call the *salted Collatz functions*, a mutation of the typical Collatz function, to erase footprints left by Collatz conjecture. In supplement, with the participation of outputs provided by this mutated Collatz function, we transform obfuscation-related predicates of Xmark's payload into mixed boolean-arithmetic (MBA) expressions (as introduced in Section II-B), so that these payload constructs could remain functionally correct after being enhanced.

Let $\theta(\cdot)$ be the typical Collatz function, and $p(n) \in \{0, 1\}$ be the parity of a natural number $n \in \mathbf{N}^*$. Assume a payload construct of Xmark where x is the condition variable involved in its underlying code obfuscation, $\phi(\cdot)$ is its control unit, and $y = \phi(x)$ is the spurious variable for manipulating its Collatz conjecture behavior. To initialize our salted Collatz function, we randomly generate the following parameters:

- a set of salt expressions $\{e_i(x) | 1 \le i \le 3\}$ which consists of n-degree polynomials of x, with coefficients of each $e_i(x)$ be even;[3] and
- coefficients $\{a_j | 1 \le j \le 6\}$ where $a_1 \div a_3 = 3$, $a_6 \div a_4 = 2$, $a_2 \equiv 0 \bmod a_3$, and $a_5 \equiv 0 \bmod a_6$.

Using this configuration, we substitutes the computing of $\theta(y)$ with Algorithm 1. Next we explain in detail the mechanism of our salted Collatz function $\vartheta(\cdot)$.

Algorithm 1 Xmark's Salted Collatz Function

Initialization: $s_0 = (a_4 e_1(x) + a_5 e_3(x))/a_6$,
$s_1 = (a_1 e_1(x) + a_2 e_2(x))/a_3 - 1$,
$y|_0 = \phi(x)$, $s|_0 = 0$, $s|_{r>0} = s_0 + s_1$ ▷

The notations $y|_r$ and $s|_r$ here indicates the value of y/s in round r of Collatz conjecture

1: **function** $\vartheta(y|_{r-1})$ ▷ assume the payload construct is at round r of its Collatz conjecture process
2: $y|_r \leftarrow y|_{r-1} + e_1(x) - s|_{r-1}$
3: **if** $y|_r \equiv 1 \mod 2$ **then**
4: $y|_r \leftarrow (a_1 y|_r + a_2 e_2(x))/a_3 + s_0$
5: **else**
6: $y|_r \leftarrow (a_4 y|_r + a_5 e_3(x))/a_6 + s_1$
7: **end if**
8: **return** $y|_r$
9: **end function**

When the assumed payload construct proceeds to run the first round of Collatz conjecture (i.e. $r = 1$), inputs of $\vartheta(\cdot)$ are respectively $y|_0 = \phi(x) = y$ and $s|_0 = 0$. Therefore, y is added with $e_1(x)$ at line 2 before further operations are computed. Since all coefficients of $e_1(x)$ are even, it's not hard to realize that $p(y + e_1(x)) = p(y) + p(e_1(x)) = p(y)$, i.e. whether $\vartheta(\cdot)$ branches to line 4 or line 6 still depends only on the parity of y (like in computing $\theta(y)$). Without loss of generality, consider the case where y is odd, the arithmetic given at line 4 of $\vartheta(\cdot)$ thus equivalents to

$$\frac{a_1(y + e_1(x)) + a_2 e_2(x)}{a_3} = 3y + 1 + \frac{a_1 e_1(x) + a_2 e_2(x)}{a_3} - 1 = \theta(y) + s_1. \tag{9}$$

Similarly, in case y is even, line 6 of $\vartheta(\cdot)$ also equivalents to $\theta(y) + s_0$. However, operations in $\theta(\cdot)$ are neither explicitly computed nor implied by any of the atomic arithmetics (or

[3] According to [33], polynomials can be considered as MBA expressions involving only addition and multiplication.

any sub-sequences of them) within the compositions at line 4/6 of $\vartheta(\cdot)$. In other words, data flow signature of Hailstone sequences cannot be detected within a single run of $\vartheta(\cdot)$.

Collatz conjecture is an iterative process, thus it's necessary to prevent the $s_{0/1}$ factor carried by the output of $\vartheta(\cdot)$ from accumulating in an uncontrollable way. Now consider the case where the assumed payload construct has proceeded to the rth round of the conjecture ($r > 1$), at which point the input $y|_{r-1}$ to $\vartheta(\cdot)$ can be decomposed into $\theta(y|_{r-2}) + s_{r-1}$. Since $\vartheta(\cdot)$ is designed to remove the s_{r-1} factor (which equals to $s_0 + s_1$) at line 2, it guarantees its later processes to work with $\theta(y|_{r-2}) + e_1(x)$, and therefore again branches exactly as if computing $\theta(y)$. Meanwhile, by adding $e_1(x)$ and removing $s_{0/1}$, signature of Hailstone sequences is effectively broken between adjacent rounds of Collatz conjecture.

As the result, we can see that while $\vartheta(\cdot)$ is constructed to be semantically inequivalent to the typical Collatz function, it is capable of producing the same branching behaviors as the latter without exposing Hailstone sequences in the software's data flow. The role of $e_1(x)$ and $s_{0/1}$ in $\vartheta(\cdot)$ is analogous to the addition of salt to a cryptographic primitive, which is the reason we named Algorithm 1 as the "salted" Collatz function. Note that although x is a variable in larger scopes, it can be safely considered as a constant between the point of entering a payload construct until exiting from it. Therefore, the concrete value of all $e_i(x)$ expressions, and consequently those of s_0 and s_1, can be determined upon initializing the salted Collatz function rather than computed within it. In addition, if treating x as a variable (even though its value won't change during the entire procedure of Collatz conjecture), then $s|_{r>0} - x = s_0 + s_1 - x$ is by all means a permutation polynomial of x (see Section II-B), thus according to Theorem 3 of [33], the inverse of $s|_{r>0} - x$ can be computed using the configuration of $\{a_j | 1 \le j \le 6\}$ and $\{m_i(x_1) | 1 \le i \le 3\}$. Let this inverse expression be denoted by s^{-1}, our enhanced payload construct determines if the Collatz conjecture loop should be terminated by evaluating "$s^{-1}(y|_r - x - 1) == x$". Similarly, (without loss of generality) assume the obfuscated predicate in our payload construct is "$x == 30$", this condition is instead checked by evaluating "$s^{-1}(y|_r - 31) == x$". The correctness of the above transformation is discussed later in Section V-B. It's also worth mentioning that line 4/6 of $\vartheta(\cdot)$ can be mutated to have diversified control structures, which would further help enhancing our payload constructs on static stealth.

Note that the sole purpose of our MBA-based transformation is to conceal static signatures of Collatz conjecture. Therefore, low-degree polynomials, like cubic or quadratic ones, should already be enough to address the requirement (while causing less overhead in the meantime). `Xmark` does not rely on MBA encoding to prohibit dynamic program analyses.

B. Watermark Recognition

There is a widely accepted generic procedure for identifying a dynamically embedded software watermark: the recognizer runs the subject software (which may or may not carry a watermark) using the secret input it possesses; these specific executions are monitored, and the recognizer collects activities (e.g. register states, control transfers and memory operations etc.) that are considered watermark-related during the process; the collected information is then interpreted according to some pre-defined rules to recover the message — or if none of such activities are found, the recognizer reports the recognition as a failure [3]. Taking into account both stealth and robustness, such a recognizer should comply to a number of assumptions to limit its capacity to a practical extent, namely:

(1) same as adversaries against software watermarking, the recognizer itself must also be assumed to have no prior knowledge on whether a subject software indeed carries a watermark or not;
(2) the recognizer should not rely on any source-code-level supportive information, like debug symbols etc.;
(3) when determining the relevance of a runtime activity to watermarking, the recognizer cannot rely on any specific static pattern of payload code, or be allowed to receive manual assistances of any kind.

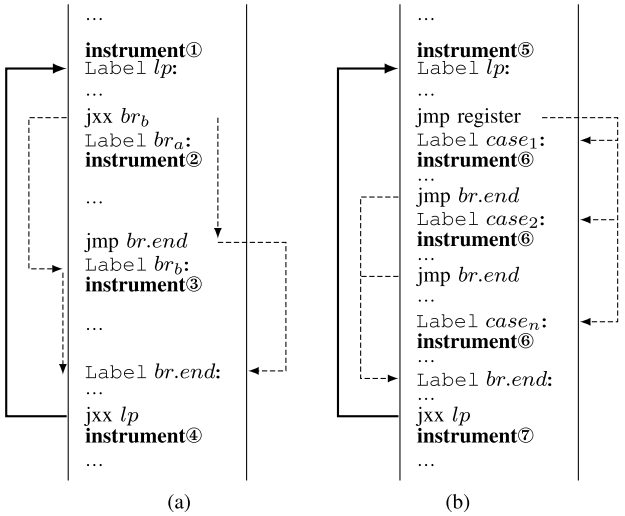


Fig. 4. Illustrative examples of `Xmark`'s loop instrumentation. (a) Basic case. (b) Control flow flattening.

The key of the aforementioned procedure is to define *what should be consider as "watermark-related"*, i.e. determining exactly what kind of activities should the recognizer be looking for. In general, the execution trace of a software is extremely noisy, where activities corresponding to a watermark (if there is one) are more or less like a needle in the sack. Specific to `Xmark`, a watermark is demonstrated by repeating a particular Hailstone sequence via control transfer activities of the Collatz function deployed within two different obfuscated conditional constructs. Due to assumption (3) given above, our recognizer cannot statically locate such payload constructs. In addition, we must also consider the scenario where adversaries try to obstruct our recognizer by altering the subject software's static structure with techniques like control flow flattening. In order to identify the watermark-related control transfers with the presence of irrelevant execution states as well as potential malicious interference, our method adopts dynamic instrumentation assisted trace analysis as the solution of the task, which consists of following operations. Without loss of generality, in the rest of this subsection, we assume it is indeed $\mathbf{S}_w$ being analyzed by our recognizer, and the watermark to be extracted is denoted by w as in all previous examples.

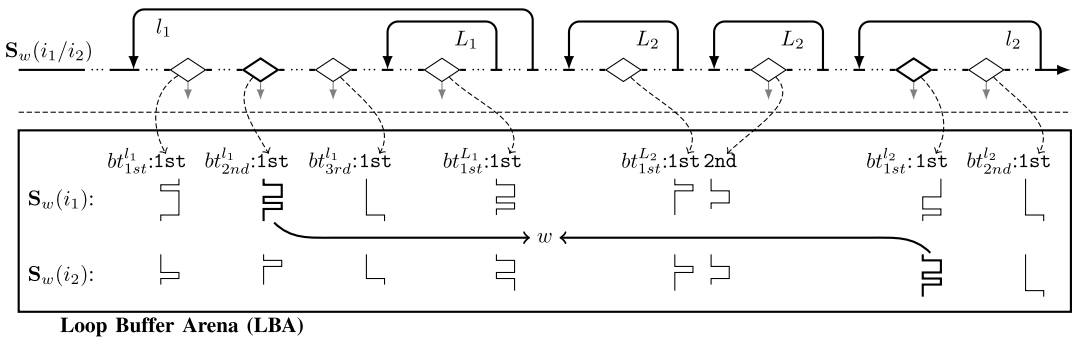


Fig. 5. An illustration of Xmark's watermark recognition.

1) Loop Instrumentation: By the mechanism of Xmark, to extract a watermark embedded by our method is in fact to track down branchings caused by the Collatz functions in our payload constructs. Let us first consider the normal case where $\mathbf{S}_w$'s control structure is not maliciously tampered with. In this case, *the Collatz function is essentially a branch structure that, along with its two conditional blocks, exist in the body of a loop* (as in Figure 2). Therefore, before actually running $\mathbf{S}_w$, Xmark examines it statically to identify all loops containing at least one conditional branch, and instruments them in the way as demonstrated in Figure 4.a. On binary level, a loop structure is typically indicated by a backward-pointing control transfer (either conditional or not), like the thick arrow in the figure from the last jxx instruction to label lp. This helps our Xmark to determine the loop's entry and exit, and consequently region of the loop body. If a loop further contains a conditional branch controlling two destination blocks (like the jxx before label br_a that points to label br_b, from which the two execution paths are given by the dashed arrows), then both destinations of the branch, together with the loop's entry and exit points, are inserted with instrument routines (e.g. **instrument①** to **instrument④** in the figure), such that

- once the loop is accessed at runtime, a variable-length buffer is created by **instrument①** for recording control transfers from the instrumented branches within it;
- when an instrumented branch is executed, **instrument②** or **instrument③** at its destinations will be reached, thus a corresponding bit is recorded into the buffer; and
- finally, after the loop breaks, the completed record is sent by **instrument④** to Xmark's recognizer.

A single loop could have multiple instrumented branches, in which case each of these branch is inserted with **instrument②** and **instrument③** in the same way. Consequently, activities from all instrumented branches in the same loop are buffered in mixture. Our recognizer parses such a mixed record into separated binary strings called the *branching trails*, which each reflects activities of a specific instrumented branch.

It is equally important for our method to be able to handle the scenario where $\mathbf{S}_w$ is maliciously re-obfuscated to make w unrecognizable. We consider control flow flattening as the adversarial tool in this case, not only because it is well studied and integrated in many obfuscation toolkits [42], [43], but also because it is lightweight enough for launching distortive attacks. This technique re-organizes normal control flow into iteratively executed switch-case structures in which basic blocks from different functionalities are mixed together. On binary level, such obfuscated control flows still exist in the form of loops. As illustrated in Figure 4.b, for a loop of this kind, Xmark attaches instrumentation routines **instrument⑤** and **instrument⑦** at its entry and exit to create and export a buffer record consists of not simple binary bits, but atomic strings of a special format encoded by **instrument⑥** within case blocks of the switch-case structure in it. Each atomic string contains the case block's distinct index and a unique ID associated with the switch-case object, which helps our recognizer to identify activities of the same flattened structure. Buffer records formed in this way are called the *raw traces*, meaning that they cannot be directly used as branching trails, but have to be further analyzed and parsed in the next step.

Intuitively, there should be many loops come and go while running a software. Therefore, Xmark's recognizer maintains a data structure called *loop buffer arena (LBA)* for all instrumented loops to create and update their buffers. With the above mechanisms, the loop instrumentation phase allows Xmark to define the exact kind of behaviors that the recognizer should focus on, and meanwhile complete all necessary preparations for extracting them from monitored executions.

2) Branching Trail Harvesting: After done instrumenting $\mathbf{S}_w$, Xmark's recognizer runs the instrumented software in a virtual machine to harvest branching trails on-the-fly. Without loss of generality, here we assume the same $\mathbf{S}_w$ described in the motivating example at the beginning of Section III, and suppose both cases of the secret input $\{i_1, i_2\}$ lead $\mathbf{S}_w$ to the same execution path. Figure 5 schematically illustrates such a path, with the backward-pointed curves indicating loops traces derived during the executions. We label loops of our payload with l_1 and l_2, and let L_1/L_2 denote irrelevant loops in $\mathbf{S}_w$. It is necessary to emphasize that a branching trail only profiles behaviors of a branch construct during one single access to the loop it is located. If a loop is accessed repeatedly,[4] like L_2 in Figure 5 (which is assumed to have been accessed twice on traces of both $\mathbf{S}_w(i_1)$ and $\mathbf{S}_w(i_2)$ in this example), a new set of branching trails are created each time. Xmark searches for echoing branching trails after the harvesting process is completed. Two branching trails are accepted as an echoing pair only if they meet a number of conditions, namely:

[4]This could have for a number of reasons, e.g. one loop residing inside another, subroutine containing the loop gets invoked repeatedly, etc.

- the trails must be respectively ported from two executions driven by contiguous cases of the secret input lineup;
- the trails must also be ported from different conditional branches of the software; and
- last but not least, the trails have to be either identical or bit-by-bit opposite to each other.

We consider the bit-by-bit opposite cases because adversaries could try to corrupt w by performing edge flipping on some branches of the software, compromising the positional relation between their destinations so that the resulting branching trails are bitwise negated. By accepting bit-by-bit opposite pairings, Xmark could reliably recognize w even if one of the echoing branches happens to be tampered with in the aforementioned way. Back to the scenario assumed in Figure 5, within the two harvested trails listed in the "Loop Buffer Arena (LBA)" area, Xmark finds that trail $[bt^{l_1}_{2nd}{:}\texttt{1st}]$ (which stands for the *first* branching trail exported by the *second* branch within loop l_1) from $\mathbf{S}_w(i_1)$ is echoed by trail $[bt^{l_2}_{1st}{:}\texttt{1st}]$ from $\mathbf{S}_w(i_2)$ (and together they indeed demonstrate w), whereas all other trails does not echo with anyone ported from the other execution. In a successful recognition, Xmark should be able to locate at least one pair of branches echoing throughout all secret-input-driven executions of $\mathbf{S}_w$. It admits branching trails ported by such branches as watermark-related.

Again, our recognizer should be able to work properly even if $\mathbf{S}_w$ is compromised using control flow flattening. Challenge brought by the re-obfuscation is that various kinds of control transfers are turned into parallel indirect jumps, therefore our recognizer can no longer use the original structure of flattened case blocks for encoding branching trails. However, since case blocks in a flattened control flow are associated with unique indexes, Xmark could infer loops and if-else branches in an execution trace using the following heuristic rules:

- intuitively, the first block of a loop is guaranteed to be executed in every round of that loop; and
- if two blocks are loop-wise mutually exclusive, i.e. each round of a loop asserts to execute one of them (but never both), these blocks indicate an if-else construct.

Therefore, given a raw trace ported from a flattened control flow (in which each unit is a case block index), an unrolled loop is in fact indicated by a group of case blocks repeating alternately. Xmark assumes the first appeared block in such a group as entry block of the loop's body, and use it to parse the unrolled loop trail into individual rounds. Our recognizer then analyzes the separated rounds to determine if there exists a loop-wise mutually exclusive block pair. In case of a while loop which evaluates loop guard in front of the loop body, its last round is omitted in this analysis. Upon identifying an exclusive block pair, Xmark randomly associates the two blocks with 0 and 1, and a binary format branching trail can therefore be organized.

3) Watermark Recovering: After extracting branching trails that are considered watermark-related, the final task for Xmark's recognizer is to recover the actual message they represent. Since the two branches of the Collatz function are mapped respectively to 0s and 1s in the branching trials, our recognizer can traverse a given trail (e.g. $[bt^{l_1}_{2nd}{:}\texttt{1st}]$ or $[bt^{l_2}_{1st}{:}\texttt{1st}]$) from its least significant bit to the most significant one, and, starting from 1, build the Hailstone sequence represented by the trail (which is $\Lambda(w)$ in this case) reversely, thus eventually recovers the initial integer of the sequence (i.e. w).

In addition, according to the rules of the Collatz function, *the odd path operation will never be the last of a Hailstone sequence, and it never appear continuously since its result is guaranteed to be even*. Hence a branching trial that indeed represents a Hailstone sequence must fit the above description (since either 0s or 1s are associated with the odd path of the Collatz function). This allows Xmark to once more validate the admitted branching trials to reduce false positives in its recognition: even an irrelevant trail accidentally survived the harvesting phase (which is already highly improbable), it can still be filtrated for having an inappropriate format.

IV. Implementation

We have implemented a prototype of Xmark based on the LLVM compiler infrastructure and the dynamic instrumentation framework Pin [44]. Currently, our embedding module works with the LLVM intermediate representation (IR) to be source- and target-agnostic, while the watermark recognizer processes x86 binaries only. In our future works, we plan to extend our method to support other instruction sets, e.g. ARM.

A. Compiler-Based Watermark Embedder

Xmark's watermark embedder is implemented as an add-on attached to LLVM's middle-end. It performs necessary analyses on the subject software's IR ported by LLVM's front-end, and based on the analysis results transforms the IR in the way as described in Section III-A to construct the watermark payload, then finally send the transformed IR to the compiler's back-end. As said in Section III-A, Xmark is capable of putting a considerably large watermark into as few as two embedding points. This allows it to be applied either on the entirety of a software, or on some selected functional modules in it. Both cases can be seen as protecting code sections belonging to a designated entry function and all subroutines reachable from the entry function (given that a software is by all means the collection of subroutines reachable from its main function). This helps Xmark determining the *subject code region* where the required embedding points should be selected. Considering the iteration of Collatz conjecture could proceed for hundreds of rounds, Xmark *avoids using conditional constructs within hot loops as embedding points* such that the resulting payload constructs would not be reached for too many times on any input case (at least for those in the software author's test suite). Our embedder analyzes static control flow graph of the subject software (provided by LLVM) to wash out branches located in condition-controlled loops. Those from count-controlled loops with small loop guards are still admissible because such loops behave in more deterministic ways. All branches in the subject code region which have survived the above screening become our candidate embedding points. Xmark then tries to generate a combination of input cases that

- each specific input case leads to an execution path that goes by at least two of the candidate points; while
- for any two neighboring input cases in the combination, their corresponding execution paths must share at least one candidate point.

Determining an input combination that fits the above criteria provides a valid secret input lineup, along with a set of actual embedding points. Xmark can then define a control unit for each of the selected branches, so that payload constructs built on them can properly derive echoing branching trails for each pair of neighboring cases of the secret input. On IR level, it is not a tough mission to find out available execution paths going through certain conditional branches and deduce possible input cases corresponding to them. Available tools on this aspect include (but not limited to) data-flow analyses, taint analyses, concolic-execution-based path verification and etc.

B. LBA as Pseudo Stacks

In binary code, a branch instruction either jumps to a remote code block (described by an offset), or simply falls through to the one right after it. These two types of destinations are often referred to as the *jump target* and the *fall-through target*, and Xmark's recognizer defines **instrument②** and **instrument③** to record a 1 if an instrumented branch directs control to its fall-through target, or a 0 otherwise. Recall that all branching trails from a single access to a loop are cached as a mixed record within one loop buffer. Built in the form of a Pintool, we let our recognizer to observe structure of loops it instrumented, and for each of them build an abstract profile to help correctly parsing the loop buffer records. To give a motivating example, consider a record 11010001 that is ported from an instrumented loop containing two conditional branches b'_1 and b'_2, with b'_2 located inside the jump target block of b'_1. Our recognizer thus generates the abstract profile for this loop, denoted by $\langle b'_1 \rightarrow \langle \texttt{jmp}, b'_2 \rangle \rangle$, then start analyzing the above loop buffer record from its most significant bit: the highest 1 indicates that b'_1 took its fall-through target in the 1st round, which means b'_2 was not reached in the same round. Therefore, the next 1 can only indicate that b'_1 again took the fall-through target in the 2nd round. As the result, eventually, the recognizer will return 11000 as the branching trail of b'_1, and in the meantime return 101 as that of b'_2. Given that the encoding format and parse rules of loop buffer record when control flow flattening is involved is already explained in Section III-B, we skip the scenario in this section.

Another problem that should be concerned is that the loop buffers are defined to be variable-length, and it is more than possible for an instrumented loop to have other instrumented loops included in its body, thus the running software may have to stop recording one loop buffer at some point to create another one. Borrowing the idea of an existing stack protection technique called SCADS [45], we let an instrumented software maintaining its LBA as pseudo stacks deployed in the unused address space between its heap and actual call stack. At the beginning of the loop instrumentation phase, our recognizer selects an *anchor address* within the available address space that is distant enough from both the allocated sections above and below it, then defines two pseudo stack structures that grow towards opposite directions from the anchor address. The layout of this structure is as demonstrated in Figure 6.a:

- the *buffer stack* stores the work-in-progress loop buffers, which is maintained using a *pseudo stack pointer* and a *pseudo base pointer*, denoted by P_{SP} and P_{BP}, with P_{SP} stored at the anchor address;
- the *counter stack* holds a group of *loop counters* created to assist the maintenance of the incomplete loop buffers, which is indexed using a *counter pointer* denoted by lc.

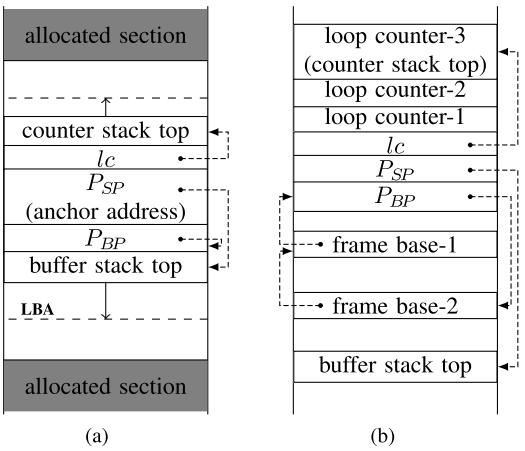


Fig. 6. Xmark's loop buffer arena built as pseudo stacks. (a) After initialized. (b) In recognition.

When an instrumented loop is accessed at runtime, the **instrument①/⑤** routine at its entry creates a new frame on top of the buffer stack, then pushes a new loop counter into the counter stack (as in Figure 6.b). This way the being examined software could flexibly adjust the size of the currently recording loop buffer, while correctly preserving all interrupted ones. When the execution breaks out from an instrumented loop, the corresponding loop buffer and loop counter should be right on top of the corresponding stacks, thus the memory space they occupy can be easily recycled by the **instrument④/⑦** routine at the exit of the loop, which is defined to unwind the two stacks after porting out the loop buffer record.

Finally, to avoid potential risk of register conflict, Xmark builds all it instrument snippets as if inline subroutines. That is, these snippets save the current register states as the first thing to be done when being executed, while recovering the saved environment as their last operation. Doing so guarantees the correctness of the subject software throughout the watermark recognition procedure, and in the meantime keeps our recognizer simple and efficient.

V. Analysis

A. Completeness

The way in which Xmark selects its embedding points has already been explained in Section IV-A. Our scheme cannot be applied if a software (or a functional program module) does not possess enough valid branches fitting the given criteria, or if its input can hardly be manipulated with external tools (e.g. when the input consists of only a boolean value). Intuitively, as long as *a software has one execution path that contains two branches and is reachable on multiple inputs*, it is a suitable subject for Xmark. Moreover, in case of last resort, it's always possible to transform part of a subject executable using tools like dynamic opaque predicates to create usable branches for watermark embedding [46]. As the result, structural detail of a program is not much of a limitation regarding

to the usability of our scheme. Having an uncontrollable input space, on the other hand, might actually make an executable inadequate to be watermarked using `Xmark`.

The recognition protocol of `Xmark` takes into account control flow level heuristics on the procedure of Collatz conjecture of two different scenarios to ensure branchings made by the Collatz function in our payload constructs be picked up during the procedure (even when the subject software's control flow is compromised via semantic-preserving transformations). It is also designed to be capable of effectively ruling out control activities from irrelevant loops using the unique characteristics of Hailstone sequences. In Section VI-D we will further verify the effectiveness of heuristics adopted be our recognizer by showing how these assumptions helps in resisting distortive attacks based on control flow flattening.

B. Correctness

1) Preventing Integer Overflow: Recall the notations of Collatz conjecture in Section II-A. Given an arbitrary $n \in \mathbf{N}^*$, it is possible to have a $k \in \mathbf{N}^*, k < \delta_n$ such that $\theta^k(n) > n$. As the result, when encoding a watermark into output of control units, `Xmark` cannot use the full length of unsigned machine integers (32/64 bits[5]). Otherwise the risk of integer overflow cannot be neglected. The main consequence of such overflow is that echoing branching activities created by our payload no longer comply to the correct Hailstone sequences, and the correctness of watermark recovery is therefore compromised. We performed an exhaustive verification and found the earliest overflow for all natural numbers under 32 bits occurred with $159,487$, i.e. the Hailstone sequence of this 18-bit number includes some intermediate values longer than 32 bits. For all natural numbers under 64 bits, the earliest overflow occurred with $12,327,829,503$ (of which the binary length is 34 bits). Therefore, by letting `Xmark` segment the watermark message into 16-/32-bit aligned substrings respectively when working with 32-/64-bit machine integers, we ensure that our payload constructs do not overflow during watermark recognition, and in the meantime achieve the best possible data rate under the respective configuration. Overflow in regular using of a watermarked software is a problem, because our payload only act the role of obfuscated control structures in this case, and the Collatz-conjecture-based obfuscation does not really care about value of the spurious variable as long as it eventually yields to 1 (which is true even if overflow occurs).

2) MBA-Based Enhancement: In Section III-A.b we have described how `Xmark` disguises data flow footprints of Collatz conjecture, which involves introducing MBA encoding into the Collatz function and obfuscation-related predicates built by our method. We have explained why this design preserves result of parity checking in the Collatz function. However, it is still necessary to demonstrate whether the transformation is indeed able to maintain the correctness of the other two key predicates in a payload construct of ours, namely the loop guard of the construct and the obfuscated conditional logic in it. Recall the example in Section III-A.b where the loop guard "$y > 1$" is replaced with "$s^{-1}(y|_r - x - 1) == x$" by our transformation, and the to-be-protected logic "$x == 30$" is obfuscated by evaluating "$s^{-1}(y|_r - 31) == x$" instead. These transformations are correct because:

- Suppose the Collatz conjecture iteration of the assumed payload construct should terminate at round k. According to Algorithm 1 our salted Collatz function would produce $y|_k = 1 + s_0 + s_1 = 1 + s|_k$, thus $y|_k - x - 1 = s|_k - x$, of which the inverse is s^{-1}. Therefore $s^{-1}(y|_r - x - 1) = x$ should hold.
- Recall that in a Collatz-conjecture-based obfuscation, the actual effective evaluation of the obfuscated conditional logic happens only when the introduced spurious variable yields to 1. Hence that during round k of the assumed payload construct, $y|_k - 31 = y|_k - x - 1$ holds only if $x = 30$. As the result, $s^{-1}(y|_k - 31) = x$ should hold only in this particular circumstance.

VI. Evaluation

We have evaluated the effectiveness `Xmark` against threats defined in the adversarial model presented in Section II-C. All test cases used in the evaluations are compiled into `x86-64` binaries using optimization option -O2. Both the watermarked instances and adversarial tools were run on a PC with an 2.7 GHz `Intel Core i5-5257U` CPU, 8 GB memory and runs a `Ubuntu` 16.04.4 LTS operation system (kernel version ubuntu-xenial 4.4.0-119-generic).

A. `Xmark` *vs. Static Pattern Matching*

In the first static-level attack to be discussed, we consider adversaries who aim to disclose location of `Xmark`'s payload constructs using pattern matching. To begin with, it's necessary to properly define the setup of such attacks in a practical and reasonable way, so that the effectiveness of our method against threats of this type can be properly evaluated.

In a real-world software, conditional branches at different positions tend to be guarding code blocks of distinct semantics (which is usually enforced by the compilers via optimizations). Therefore, the similarity between any two payload constructs of `Xmark` is intrinsically low, given that they carry very different code blocks guarded by their embedding points. We assume that our adversaries are aware of the fact that `Xmark` disguises its payload with MBA encoding, but according to assumptions stated in Section II-C, detailed implementation of the enhancement, e.g. the value of parameters generated in the random initialization or the concrete control structure of the salted Collatz function, remains a secret to any third-party. Taking these into account, the best adversarial strategy we can think of is to look for patterns of the typical Collatz function (the only possible artificial signature of our payload) with the presence of MBA-based enhancement. Thus in this evaluation, such a typical sample of Collatz-conjecture-based obfuscation routine was adopted as the adversarial template, and was then compared with a collection of various algorithms to show if the comparison could single out implementations of Collatz conjecture deployed in these algorithms with high significance. In addition, we build the adversarial template to be as simple as

[5]We want `Xmark` to be downward compatible so that old 32-bit software can also be protected.

TABLE II

DESCRIPTIONS OF ALGORITHMS USED AS THE CONTROL GROUP OF OUR STEALTH SIMULATION

Algorithm	Abbrev	Compared portion
Minimum Spanning Tree	MST	main operation
Depth First Search	DFS	main operation
Greedy Algorithm	GA	array operation
Quick Search	QS	main operation
Rabin-Karp Algorithm	RKA	whole algorithm
Native Pattern Search	NPS	whole algorithm

possible, i.e. the obfuscated conditional construct only guards a simple arithmetic operation like "i++", to make the Collatz function within this template dominating the matching.

1) Test Groups: We divide the algorithms to be compared with our adversarial template into two groups. The *experiment group* consisted of subject routines that were indeed payload constructs built by `Xmark`. To clarify the effect of MBA-based enhancement, we further classifies these instances into 3 types:

a. obfuscation in these instances involved no enhancement, but their subject conditional constructs were individually different (with the adversarial template excluded);
b. these instances obfuscated the same conditional construct used in the adversarial template, but were also enhanced with MBA encoding implemented in randomized ways;
c. these instances simulated the practical scenario, i.e. they were built by obfuscating different conditional constructs and applying MBA-based enhancement in the same time.

The second group was our *control group*, containing a number of classical algorithms which have nothing to do with Collatz conjecture (Table II have listed the specific algorithms selected for this group). The philosophy behind this simulation is: if the assumed adversaries could indeed obtain advantage via pattern matching, similarity between the adversarial template and the experiment group samples should be significantly higher than that between the adversarial template and algorithms from the control group. Otherwise, if two groups of comparisons failed to demonstrate significant differences, then it should be safe to consider `Xmark` as stealthy against attacks of this type.

2) Simulation Results: our simulation is carried out using two binary diffing tools, namely `Radare2`[6] and `Bindiff`.[7] Both tools adopt graph- or structure-based comparison strategy to provide function-wise binary comparison[8] [47], [48]. Note that ideally, our simulation should be looking for small code pieces similar with the adversarial template in a much larger code region. We had to settle for function-wise comparisons because we could not find a widely accepted analysis tool of which the functionality meets this ideal scenario. In addition, to the best of our knowledge, `Radare2` and `Bindiff` adopts the most appropriate similarity metrics for pattern matching `Xmark`'s payload constructs, considering the implementation of Collatz conjecture gives no other static features like unusual instructions or specific system/API calls. Figure 7 presents a few examples from both groups of the comparisons ported by `Bindiff`, with the left part of each comparison be our adversarial template, and their right part be instances selected from the two groups described above. Table III also presents full results of this simulation. Overall, similarities between the adversarial template and instances of the experiment group varied from 0.113/0.27 to 0.395/0.76 (`Radare2`/`Bindiff`), while those between the adversarial template and control group algorithms were in the range of 0.108/0.32 and 0.28/0.72. This suggests that with code similarity as the metric, adversaries cannot reliably distinguish which specific group does a being compared program belong to. Furthermore, similarity results on different types of experiment instances suggests that MBA-based enhancement contributed significantly in impeding pattern matching. Comparisons between the adversarial template and type-b experiment instances showed lower similarities than those with the type-a experimental instances, even though the former group were essentially the same conditional construct on which the adversarial template itself is built on. Finally, when facing the type-c experiment instances, both `Bindiff` and `Radare2` reported the worst similarity result in all simulations. Together, these results suggest that using the assumed pattern matching attack, adversaries can neither determine if a subject executable is actually watermarked via `Xmark`, nor locate the embedded payload constructs.

TABLE III

RESULTS OF OUR STEALTH SIMULATION (CODE SIMILARITY COMPARISON) AGAINST `Xmark`

Group	Case	Comparison result	
		`Radare2`	`Bindiff`
Experimental Group	type-a	0.266	0.500
		0.395	0.760
	type-b	0.273	0.720
		0.308	0.690
	type-c	0.113	0.270
		0.132	0.270
Control Group	MST	0.122	0.460
	DFS	0.183	0.720
	GA	0.280	0.720
	QS	0.097	0.320
	RKA	0.108	0.450
	NPS	0.182	0.580

B. `Xmark` *vs. Model Checking*

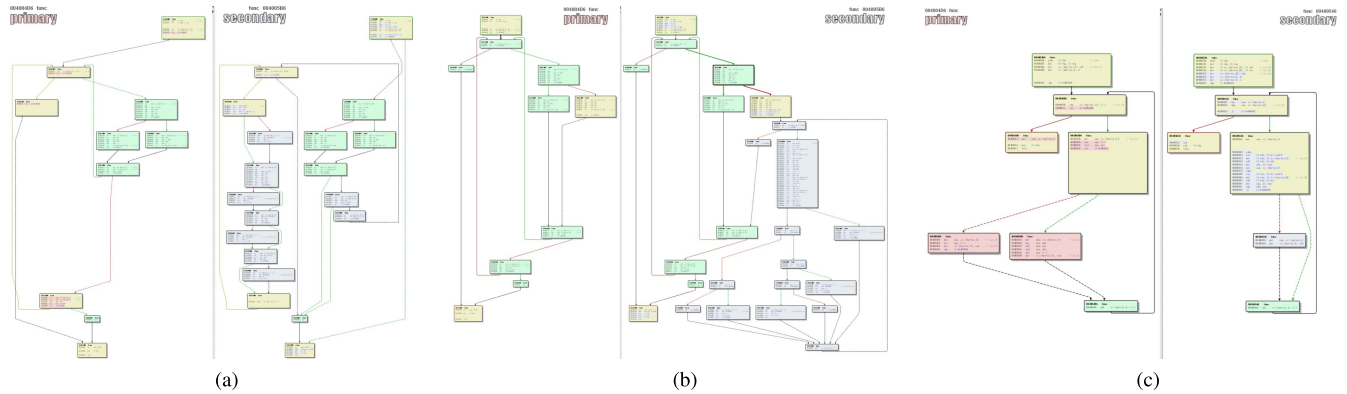
A more sophisticated static-level attack to be considered is when adversaries leverage model checking to verify whether a subject software contains payload of `Xmark`. As stated in Section II-C, we assume that this attack uses model checking in the way as for malware detection purpose, where the model of a program is checked against specifications of a target code (i.e. signatures describing known behavior of the target) [37]. Instead of targeting malware, adversaries against `Xmark` could build specifications for the routine of Collatz conjecture, then move to infer the existence as well as location of our payload constructs by applying a model checking as described above to verify a subject software against their specifications.

A successful model checking relies heavily on the accuracy of target specifications. For attacks against `Xmark`, however, the only priori knowledge for the adversaries to establish such specifications is again the typical routine of Collatz conjecture (same as in the previously discussed pattern matching attack). However, due to the existence of MBA-based enhancement,

[6] http://www.radare.org/r/

[7] https://www.zynamics.com/bindiff.html

[8] Because of this, all subject algorithms, including the adversarial template and instances of both test groups, are implemented as independent functions.



(a) (b) (c)

Fig. 7. Examples of the code comparison between the adversarial sample and the test cases. (a) Versus type-a experimental instance. (b) Versus type-b experimental instance. (c) Versus control group instance (GA).

our payload constructs are mounted with the salted Collatz function that is semantically mutated from its origin. As the result, the specifications which adversaries abstracted out of the typical Collatz conjecture routine would be missing too much vital information regarding to the actual implementation of `Xmark`. Without reliable specifications of the target program, chance for model-checking-based attacks to successfully disclose our payload constructs is slim.

C. `Xmark` *vs. Dynamic Program Analyses*

On dynamic level, we assume two different attacks against `Xmark`. First off, leveraging program testing techniques like fuzzing and etc., adversaries could randomly probe a subject software's execution paths, hoping to trigger some watermark-related activities by accident (and consequently disclose the location corresponding payload code). Since our recognition protocol can be learnt publicly, adversaries could build their own recognizers that is equally effective in catching behaviors of Collatz conjecture. However, according to the mechanism of `Xmark`, *an accidental exposure of any segments of an embedded watermark would happen only if the watermarked software is run on neighboring cases of the secret input in the correct order*. In practice, a software's input could consist of anything from a configuration value to a mouse click on a GUI object, making it an unlikely event to encounter such kind of "lucky draws" via random program testing.

In the next attack scenario, adversaries could try to compromise or remove the payload constructs of `Xmark` using more sophisticated analysis techniques such as symbolic execution, constraint solving and etc. This attack could serve as a follow-up once some components of our payload have been exposed by the probing attack. It could also fly solo to indiscriminately de-obfuscates the entire subject software. `Xmark` is integrated with the Collatz-conjecture-based obfuscation technique which is said to be a special type of dynamic opaque predicate [29]. The obfuscation specifically exploits an intrinsic shortcoming of symbolic execution, i.e. state explosion in loop unrolling, making it (and `Xmark` which is built on it) much more difficult to be solved and removed compare to existing methods which adopted invariant opaque predicates [7], [18].

Our second simulation aims to evaluate what happens when adversaries choose to analyze a payload construct of `Xmark` using symbolic execution. Note that in such a scenario, it is the obfuscation nature of our payload standing in the way of the program analyzers. In order to focus completely on how the obfuscation affects this reverse engineering technique, this simulation made the assumed adversaries examine a subject software consisting of nothing else but a simple obfuscation routine without applying MBA encoding, so that contextual disturbances which adversaries have to face is minimized. We used the same instance given in Figure 2c of [31] (in which the obfuscated predicate is "$x == 30$") as the sample to be analyzed, while `angr`, a recently proposed binary analysis framework, was used as the adversarial tool [49]. We set up this simulation in such a way because with the improvement of SMT solver, symbolic execution has become more powerful than a decade ago, making it necessary to study whether the state-of-the-art analyzer has now overturned past conclusions regarding to the Collatz-conjecture-based obfuscation. Also, `angr` is an open source framework, which makes it possible to closely observe exactly how the sample instance is processed.

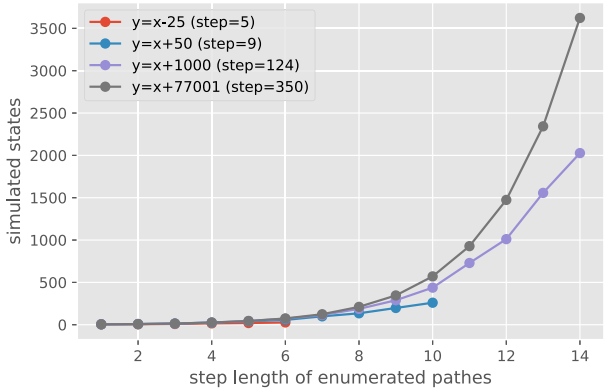


Fig. 8. Growth rate of `angr`'s simulated states.

Our first observation is that exploring a conditional construct obfuscated using the Collatz-conjecture-based approach is not always equally difficult. Trivial instances do exist which can be solved rather easily. For example, let the spurious variable be computed by $y = x - 25$, i.e. when the obfuscated condition satisfies, our obfuscated object is initiated by $y = 5$ (and the corresponding Hailstone sequence has only 6 steps), `angr` solved this specific case in the matter of seconds. After looking into `angr`'s log in detail, we found that on the first pass of

our obfuscated construct, angr considered it as symbolic and started path exploration while unrolling the Collatz conjecture loop. Since the conjecture is not a theorem, there was no way for angr to understand in advance that the explored routine always terminates when $y = 1$. As the result, the exploration of this analyzer worked according to the following strategy:

(1) according to the definition of Collatz function, enumerate all valid paths after running the explored instance for a specified number of steps (starting from 1);
(2) for each enumerated path, use constraint solving to see if the obfuscated condition logic is already satisfiable, and return the solution if so;
(3) if otherwise, increase the specified step length by 1, then return to step (1).

The above procedure would keep iterating until correct trigger of the obfuscated branch is found eventually. This observation suggests that unless the execution path corresponding to the being explored trigger condition is short enough, angr will soon face path explosion during analyzing our payload objects. To verify this argument, we modified the generator of spurious variable in our sample instance into $y = x + ag$, and set ag respectively to 50, 1000 and 77001. Put in other words, when the obfuscated predicate is satisfied, these modified instances would accordingly be controlled to walk 9, 124 and 350 steps until the obfuscated branches in them were satisfied. Figure 8 shows the growth rate of simulated program states maintained by angr when all four instances mentioned in this subsection were analyzed. We can see that angr did successfully solve the two trivial cases. But when processing the other two, the simulated states it had to maintain rapidly increased to over 2000/3500 only after enumerating possible Collatz conjecture paths for 14 steps. This burst of states crashed angr after prolonged yet fruitless analyses, and before any path of 15 steps was able to be enumerated. Recall that the obfuscation constructs used this simulation were not yet enhanced using MBA encoding. Thus we can safely assume that when set in its full configuration, Xmark can prohibit dynamic analyses driven by symbolic execution even more effectively than what is shown in this simulation. More importantly, results of this simulation suggest that even if a probing attack managed to detect some payload constructs of our method, it would still be a difficult task to further compromise or remove them.

D. Xmark *vs. Control Flow Flattening*

One last attack scenario to be discussed involves adversaries who attempt to distort a watermark embedded using Xmark by indiscriminately manipulating static structure of a subject software using control flow flattening. As said in Section III-B and V-A, Xmark adopts heuristics on control flow behavior of Collatz conjecture to collect watermark-related program states, which covers payload constructs re-obfuscated by control flow flattening. We performed our third simulation to demonstrate how this feature helps our method resisting such attacks. In this simulation, we extended the simple obfuscation routine used previously in Section VI-C into a more realistic instance by applying MBA-based enhancement on it. Control flow of this disguised test instance is given in the left half of Figure 9. Our transformation introduced an extra branch into the even path of Collatz function within the test instance in order to provide a better picture on how Xmark's recognizer work on payload constructs with diversified structures. Re-obfuscated version of the test instant is built using ollvm, a well-known obfuscation tool able to perform control flow flattening [43]. The flattened control flow of our test instance is given in the right half of Figure 9. All code blocks in the figure are indexed as how they were labeled by LLVM. We executed both test instances with their Collatz conjecture loop controlled by $y = 3$ to evaluate the resilience of Xmark against control flow flattening.

Execution trace of the two test instances (in the form of basic block sequences) are presented in the middle of Figure 9. Recall that as said in Section III-B, normally Xmark focus on branches within loop bodies, while for looped switch-case structures it only instruments the case blocks. Therefore, trace of the disguised-only instance consisted of blocks denoted by dark squares, where block 16 and 20 (marked by "[]" and "()" respectively) were the two operations of Collatz function. We consider the disguised-only instance as the baseline case. On the other hand, trace of the flattened instance only consisted of block 12 to 52 in the right half of Figure 9. According to heuristics as also given in Section III-B, our recognizer identified block 13 as the first appeared repeating block (which is marked with "⟨⟩"), and use it to parse the flattened trace into individual loop rounds. It is then easy to see that after omitting the last round, block 22 and 26 formed a loop-wise exclusive pairing, the sequence of which was accepted by our recognizer as a branching trail. After compared with the baseline case, we can see that block 22 and 26 in the flattened trace in fact correspond to block 16 and 20 in the baseline trace, suggesting that Xmark had correctly resolved the same branching trail from the compromised control flow. We believe this result has demonstrated that Xmark's recognition protocol is capable of resisting structure level distortive attacks like control flow flattening, which also supported our completeness argument as given in Section V-A.

E. Performance

There is a silent consensus that performance of a software watermarking method is mainly about its impact to the subject software caused by the watermark-related modifications, while the watermark embedder and recognizer themselves are more often than not deemed as off-line processes. Evidence of such an assumption can be found in many existing works from how these schemes were evaluated on this aspect [15], [16], [19], [23]. In this paper, we have evaluated Xmark's performance according to the same philosophy.

We generated a total of 10 random 128-bit binary strings as assumed watermarks, which were then embedded into selected SPEC-CINT-2006 benchmarks using Xmark. Our method managed in watermarking all 12 benchmarks included in the test suite. However, we found that for 6 of the benchmarks, all valid embedding points we identified cannot be reached using the official workload of SPEC (see Section 3.3 of [50]). Due to this observation, we chose to omit these benchmarks in our evaluation, since otherwise they would only make the

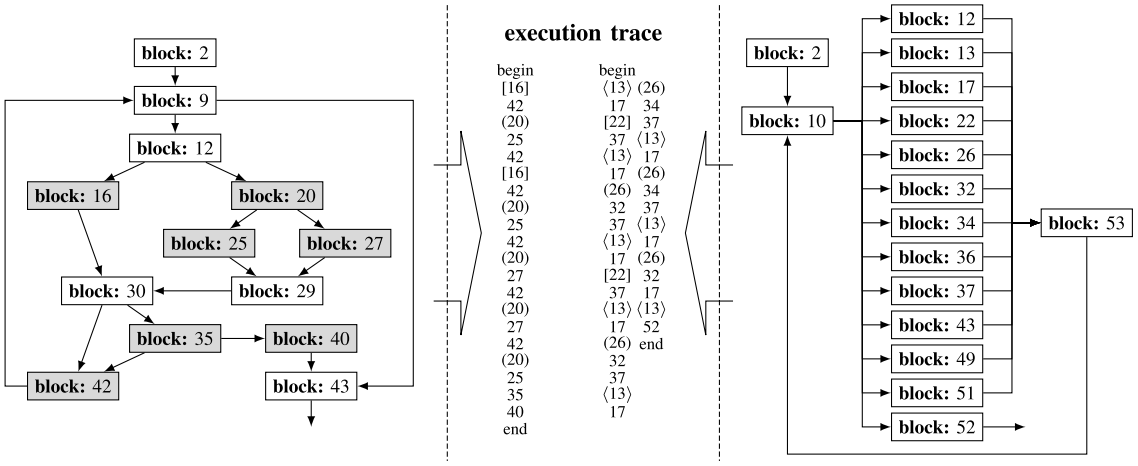


Fig. 9. Simulation of Xmark's recognition process with & without control flow flattening.

TABLE IV
CODE BLOAT AND PERFORMANCE OVERHEAD CAUSED BY Xmark

Benchmark	Target Function	Control group		Xmark			
		target function size(bytes)	avg time consuming(s)	target function size(bytes)	code bloat(bytes)	avg time consuming(s)	performance overhead(%)
perlbench	perl_destruct, S_parse_body	1190	419.936	1798	608	418.873	-0.253
gcc	init_caller_save	7172	294.931	8211	1039	294.809	-0.041
hmmer	SSIOpen, HMMFileOpen	11159	387.471	11711	552	387.628	+0.041
gobmk	main	7112	508.985	7716	604	509.229	+0.048
astar	regmngobj::loadmap, wayobj::loadmap	3024	410.825	3761	737	412.370	+0.376
h264ref	Configue	1061	546.174	1658	597	545.598	-0.105

results unfairly better. We described the performance impact of Xmark by measuring code bloat and runtime overhead of the watermarked projects. Code bloat was assessed simply by comparing size of executables before and after watermarking. We implemented the control units of our payload constructs in the form of segmented functions, so that embedding different watermarks only changes size of the payload in a limited way (which strengthens the credibility of our evaluation results). To demonstrate runtime overhead induced by our watermarking, we put each of the original and the watermarked benchmarks though 5 rounds of tests, with each of these tests complying to SPEC's standard procedure. Slowdown of the watermarked benchmarks was then measured by comparing the average time consuming of the two groups of tested executions.

We found that when analyzing the benchmarks as executable files, the overall size increments caused by watermarking was always of a number either less than ± 200 bytes or close of 4KB (size of a memory page). This suggested that padding and alignment done by the file system contributed too much in those increments compared with watermark embedding. To reach more accurate results, we used IDA pro to measure the exact size of target functions (where Xmark deployed the payload constructs) in each evaluated benchmark, to see the actual increment caused by the embedding. From Table IV we can see that the embedding transformation of Xmark caused an average code bloat of around 460 bytes to the residential code modules of its payload, with the worst case obverted in the tests a little bit more than 1KB. Results on overhead further showed that in all the tested cases, slowdown introduced by Xmark's watermark payload was not significant.

It is necessary to understand that code bloat and overhead caused by Xmark's watermark payload depend heavily on the implementation of its control units and MBA-based disguising. Also, in its worst cases, a Collatz conjecture routine could iterate for hundreds of round before terminating, and overhead caused to the corresponding embedding point could increase severalfold compared to the average scenarios. Our simulation provided an empirical estimation on the average situation of Xmark's performance. Nevertheless, we admit that results given in this section does not cover the worst scenario.

VII. CONCLUSION

We have presented a novel dynamic software watermarking scheme name Xmark, which is built on the Collatz-conjecture-based control flow obfuscation. Our method exploits Hailstone sequences to encode binary messages into iterative branching behaviors produced in computing Collatz conjecture, while making a so encoded watermark recognizable by manipulating different Collatz conjecture routines to walk through the same Hailstone sequence on different execution traces. This mechanism, together with the MBA-based enhancement applied to its payload constructs, makes Xmark able to conceal watermark-related code and activities from most unauthorized external observations. The obfuscation nature of Xmark's payload also makes it functionally emerged with the subject software. Our method overcomes a series of known weaknesses existed in

conventional dynamic watermarking solutions. It is capable of carrying large watermark with light-weight payload, and could effectively resist targeted attacks based on a number of static and/or dynamic program analysis techniques including pattern matching, model checking and symbolic execution. Moreover, as a control flow based scheme, our method is robust against distortive attacks using control flow flattening or edge flipping.

REFERENCES

[1] The Software Alliance. (2018). *Software Management: Security Imperative, Business Opportunity*. [Online]. Available: https://gss.bsa.org/wp-content/uploads/2018/05/-2018_BSA_GSS_Report_en.pdf

[2] W. F. Zhu, "Concepts and techniques in software watermarking and obfuscation," Ph.D. dissertation, Dept. Comput. Sci., Univ. Auckland, Auckland, New Zealand, 2007.

[3] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, 1st ed. Reading, MA, USA: Addison-Wesley, 2009.

[4] A. Dey, S. Bhattacharya, and N. Chaki, "Software watermarking: Progress and challenges," *INAE Lett.*, vol. 4, no. 1, pp. 65–75, 2018.

[5] R. I. Davidson and N. Myhrvold, "Method and system for generating and auditing a signature for a computer program," U.S. Patent 5 559 884, Sep. 24, 1996.

[6] R. Venkatesan, V. V. Vazirani, and S. Sinha, "A graph theoretic approach to software watermarking," in *Proc. 4th Int. Workshop Inf. Hiding*, 2001, pp. 157–168.

[7] G. Arboit, "A method for watermarking java programs via opaque predicates," in *Proc. 5th Int. Conf. Electron. Commerce Res.*, 2002, pp. 102–110.

[8] R. El-Khalil and A. D. Keromytis, "Hydan: Hiding information in program binaries," in *Proc. Int. Conf. Inf. Commun. Secur.*, 2004, pp. 187–199.

[9] C. Collberg and T. R. Sahoo, "Software watermarking in the frequency domain: Implementation, analysis, and attacks," *J. Comput. Secur.*, vol. 13, no. 5, pp. 721–755, 2005.

[10] W. Zhu and C. Thomborson, "Extraction in software watermarking," in *Proc. 8th workshop Multimedia Secur.*, 2006, pp. 175–181.

[11] H. Lee and K. Kaneko, "New approaches for software watermarking by register allocation," in *Proc. 9th ACIS Int. Conf. Softw. Eng., Artif. Intell., Netw., Parallel/Distrib. Comput.*, 2008, pp. 63–68.

[12] D. Gong, F. Liu, B. Lu, P. Wang, and L. Ding, "Hiding informationin in Java class file," in *Proc. Int. Symp. Comput. Sci. Comput. Technol.*, 2008, pp. 160–164.

[13] J. Hamilton and S. Danicic, "A survey of static software watermarking," in *Proc. World Congr. Internet Security*, 2011, pp. 100–107.

[14] C. Collberg and C. Thomborson, "Software watermarking: Models and dynamic embeddings," in *Proc. 26th ACM SIGPLAN-SIGACT Symp. Principles Program. Lang.*, 1999, pp. 311–324.

[15] C. Collberg *et al.*, "Dynamic path-based software watermarking," in *Proc. ACM SIGPLAN Conf. Program. Language Design Implement.*, 2004, pp. 107–118.

[16] J. Nagra and C. Thomborson, "Threading software watermarks," in *Proc. 6th Int. Workshop Inf. Hiding*, 2005, pp. 208–223.

[17] G. Myles and H. Jin, "Self-validating branch-based software watermarking," in *Proc. Int. Workshop Inf. Hiding*, 2005, pp. 342–356.

[18] G. Myles and C. Collberg, "Software watermarking via opaque predicates: Implementation, analysis, and attacks," *Electron. Commerce Res.*, vol. 6, no. 2, pp. 155–171, 2006.

[19] C. S. Collberg, C. Thomborson, and G. M. Townsend, "Dynamic graph-based software fingerprinting," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 6, 2007, Art. no. 35.

[20] Y. Ke-Xin, Y. Ke, and Z. Jian-Qi, "A robust dynamic software watermarking," in *Proc. Int. Conf. Inf. Technol. Comput. Sci.*, vol. 1, 2009, pp. 15–18.

[21] X. Zhang, F. He, and W. Zuo, "Hash function based software watermarking," in *Proc. Int. Workshop Digit. Watermarking Advanced Softw. Eng. Appl.*, 2008, pp. 95–98.

[22] W. Zhou, X. Zhang, and X. Jiang, "AppInk: Watermarking Android apps for repackaging deterrence," in *Proc. 8th ACM SIGSAC Symp. Inf., Comput. Commun. Secur.*, 2013, pp. 1–12.

[23] C. Ren, K. Chen, and P. Liu, "Droidmarking: Resilient software watermarking for impeding Android application repackaging," in *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng.*, Jan. 2014, pp. 635–646.

[24] H. Ma, K. Lu, X. Ma, H. Zhang, C. Jia, and D. Gao, "Software watermarking using return-oriented programming," in *Proc. 10th ACM Symp. Inf., Comput. Commun. Secur.*, 2015, pp. 369–380.

[25] H. Ma, R. Li, X. Yu, C. Jia, and D. Gao, "Integrated software fingerprinting via neural-network-based control flow obfuscation," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 10, pp. 2322–2337, Oct. 2016.

[26] M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere, "Hybrid static-dynamic attacks against software protection mechanisms," in *Proc. 5th ACM Workshop Digit. Rights Manage.*, 2005, pp. 75–82.

[27] G. Gupta and J. Pieprzyk, "A low-cost attack on branch-based software watermarking schemes," in *Proc. Int. Workshop Digit. Watermarking*, 2006, pp. 282–293.

[28] M. D. Preda, M. Madou, K. De Bosschere, and R. Giacobazzi, "Opaque predicates detection by abstract interpretation," in *Proc. Int. Conf. Algebraic Methodol. Softw. Technol.*, 2006, pp. 81–95.

[29] J. Ming, D. Xu, L. Wang, and D. Wu, "LOOP: Logic-oriented opaque predicate detection in obfuscated binary code," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2015, pp. 757–768.

[30] J. Lopez, L. Babun, H. Aksu, and A. S. Uluagac, "A survey on function and system call hooking approaches," *J. Hardw. Syst. Secur.*, vol. 1, no. 2, pp. 114–136, 2017.

[31] Z. Wang, J. Ming, C. Jia, and D. Gao, "Linear obfuscation to combat symbolic execution," in *Proc. 16th Eur. Symp. Res. Comput. Secur.*, 2011, pp. 210–226.

[32] J. C. Lagarias, *The Ultimate Challenge: The 3×1 Problem*. Providence, RI, USA: AMS, 2010.

[33] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson, "Information hiding in software with mixed Boolean-arithmetic transforms," in *Information Security Applications*. Berlin, Germany: Springer, 2007, pp. 61–75.

[34] R. L. Rivest, "Permutation polynomials modulo 2^w," *Finite Fields Appl.*, vol. 7, no. 2, pp. 287–292, 2001.

[35] G. Danezis, "Statistical disclosure attacks," in *Proc. IFIP Int. Inf. Secur. Conf.* Boston, MA, USA: Springer, 2003, pp. 421–426.

[36] K. Harrison and S. Xu, "Protecting cryptographic keys from memory disclosure attacks," in *Proc. 37th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2007, pp. 137–143.

[37] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Proactive detection of computer worms using model checking," *IEEE Trans. Dependable Secure Comput.*, vol. 7, no. 4, pp. 424–438, Oct. 2010.

[38] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of software-based survivability mechanisms," in *Proc. 31th Int. Conf. Dependable Syst. Netw.*, 2001, pp. 193–202.

[39] F. Cohen, "Computer viruses: Theory and experiments," *Comput. Secur.*, vol. 6, no. 1, pp. 22–35, 1987.

[40] F. Hohl, "Time limited blackbox security: Protecting mobile agents from malicious hosts," in *Mobile Agents and Security*. Berlin, Germany: Springer, 1998, pp. 92–113.

[41] Z. Wu, S. Gianvecchio, M. Xie, and H. Wang, "Mimimorphism: A new approach to binary code obfuscation," in *Proc. 17th ACM Conf. Comput. Commun. Secur.*, 2010, pp. 536–546.

[42] C. Collberg, S. Martin, J. Myers, and J. Nagra, "Distributed application tamper detection via continuous software updates," in *Proc. 28th Annu. Comput. Secur. Appl. Conf.*, 2012, pp. 319–328.

[43] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM—Software protection for the masses," in *Proc. IEEE/ACM 1st Int. Workshop Softw. Protection*, May 2015, pp. 3–9.

[44] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2005, pp. 190–200.

[45] C. Kugler and T. Müller, "Separated control and data stacks to mitigate buffer overflow exploits," *EAI Endorsed Trans. Secur. Saf.*, vol. 2, no. 4, pp. 1–34, 2015.

[46] D. Xu, J. Ming, and D. Wu, "Generalized dynamic opaque predicates: A new control flow obfuscation method," in *Proc. Int. Conf. Inf. Secur.*, 2016, pp. 323–342.

[47] T. Dullien and R. Rolf, "Graph-based comparison of executable objects (English version)," in *Proc. Symp. sur la Securite des Technol. de l'Inf.et des Commun.*, 2005, p. 3.

[48] H. Flake, "Structural comparison of executable objects," in *Proc. Int. GI Workshop Detection Intrusions Malware Vulnerability Assessment*, 2004, pp. 161–174.

[49] Y. Shoshitaishvili *et al.*, "SOK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy*, May 2016, pp. 138–157.

[50] SPEC-Open-Systems-Group. (2011). *SPEC CPU2006 Run and Reporting Rules*. [Online]. Available: https://www.spec.org/cpu2006/Docs/runrules.html

Authors' photographs and biographies not available at the time of publication.