SoftMark: Software Watermarking via a Binary Function Relocation

Honggoo Kang
Korea University
Seoul, South Korea
honggoonin@korea.ac.kr

Sangjin Lee
Korea University
Seoul, South Korea
sangjin@korea.ac.kr

Yonghwi Kwon
University of Virginia
Charlottesville, Virginia, USA
yongkwon@virginia.edu

Hyungjoon Koo∗
Sungkyunkwan University
Suwon, South Korea
kevin.koo@skku.edu

ABSTRACT

The ease of reproducibility of digital artifacts raises a growing concern in copyright infringement; in particular, for a software product. Software watermarking is one of the promising techniques to verify the owner of licensed software by embedding a digital fingerprint. Developing an ideal software watermark scheme is challenging because i) unlike digital media watermarking, software watermarking must preserve the original code semantics after inserting software watermark, and ii) it requires well-balanced properties of credibility, resiliency, capacity, imperceptibility, and efficiency. We present SoftMark, a software watermarking system that leverages a function relocation where the order of functions implicitly encodes a hidden identifier. By design, SoftMark does not introduce additional structures (i.e., codes, blocks, or subroutines), being robust in unauthorized detection, while maintaining a negligible performance overhead and reasonable capacity. With various strategies against viable attacks (i.e., static binary re-instrumentation), we tackle the limitations of previous reordering-based approaches. Our empirical results demonstrate the practicality and effectiveness by successful embedding and extraction of various watermark values.

CCS CONCEPTS

• Security and privacy → Software security engineering.

KEYWORDS

Software Watermarking, Watermark, Function Reordering, Function Relocation, Binary Instrumentation

1 INTRODUCTION

Today, a vast usage of digital data makes our life convenient by sharing them with others due to its trivial reproducibility by nature. However, the ease of both data duplication and distribution raises unfavorable consequences, that is, copyright infringement when digital contents (e.g., pictures, movies, TV episodes, software) are illegally copied, distributed, or publicly presented without the owner’s permission. The number of disputes over copyrights on pirated materials gradually increases; in particular, software piracy is a significantly growing concern. According to the survey conducted by BSA [10], 37% of the whole software around the globe have been estimated as illegitimate or unlicensed with the commercial value of $46.3 billion.

Digital watermarking is one of promising techniques for recognizing the originality of digital works. It covertly inserts a unique digital fingerprint into digital contents such as text, image, audio, and video so that the ownership of the contents can be identified by revealing the embedded fingerprint. In a similar vein, software watermarking is a technique that aims to provide the digital fingerprint of a software product by inserting certain information that represents its owner or distributor. Then, the identifier of every software copy offers the traceability because it belongs to a unique customer upon the purchase of software.

Software watermarking is effective against an adversary who wants to run a copyrighted program free of charge, revealing neither the identity of the attacker nor the original owner of the program. The adversary may attempt to reverse-engineer a watermark embedding process as well as unauthorized detection. While it is nearly impossible to achieve complete prevention against all viable attacks, a desirable software watermarking scheme should be able to provide a sufficient level of stealthiness and resiliency that renders such attacks extremely expensive, or severely discourages attackers. To this end, as with previous work [14, 21, 23, 27, 42, 49, 67], we identify key properties (requirements) of software watermarking techniques: Credibility, Capacity, Imperceptibility, Resiliency, Spread, and Efficiency (See §2 in detail).

For the last few decades, diverse software watermarking [23, 28, 70] approaches have been proposed, including reordering-based [22, 30, 51, 53, 55], graph-based [14, 18, 32, 48, 49, 69], obfuscated-based [4, 5, 11, 15, 33, 41, 66], and branch-based [26] approaches. Depending on where/how the watermark is inserted and verified, software watermarking techniques in the literature
can be classified into static [4, 22, 30, 32, 41, 48, 49, 51, 53, 55] or dynamic [14, 18] approaches. A static watermarking technique does not need to run a program whereas a dynamic watermarking technique extracts a watermark at runtime. Unfortunately, we observe that the existing approaches have difficulty in achieving desirable balances between the properties, particularly resiliency and imperceptibility.

In this paper, we propose SoftMark, a software watermarking technique on top of a function relocation scheme. Since reliable relocation of binary functions is extremely challenging, our technique is highly resilient against varying attacks including unauthorized detection, illegal corruption and collusion. Moreover, SoftMark is implicitly encoded, leveraging the location of pre-selected functions in a target program; each order of the functions maps into a secret identifier.

Our watermarking scheme has addressed several drawbacks of previous reordering based approaches [22, 42] by adopting fruitful strategies that impose significant challenges to watermarking corruption techniques via static binary instrumentation. First, SoftMark does not introduce any codes, blocks, or subroutines to a target program, which empirically demonstrates negligible runtime and space overheads. Second, the presence of a watermark is difficult to reveal by a statistical analysis or inference unless multiple instances are collusively collected. Third, SoftMark conveys a relatively high capacity for watermark encoding, which is proportional to the size of a program (i.e., A set of n functions can represent up to $\log_2 n!$ bits). Fourth, the design of SoftMark shows a reasonable resiliency even under semantic-preserving code transformation by inserting multiple watermarks across a broad spectrum of functions. It is noteworthy mentioning that we select a set of unique functions with a variety of strategies that make reliable code transformation challenging. To implement SoftMark, we employ CCR [35], a special compiler toolchain that emits metadata for instrumenting a variant with a watermark.

In summary, we make the following contributions:

- We propose SoftMark, an efficient watermarking system via a function relocation based encoding, resolving most of the prior limitations.
- We have designed and implemented a prototype of SoftMark to meet the requirements of a practical software watermarking technique against various viable attacks.
- We experimentally evaluate SoftMark with real world applications, demonstrating the effectiveness and practicality of our approach.

The source code of SoftMark will be publicly available in the near future to foster further watermarking research.

2 SOFTWARE WATERMARKING

In this section, we discuss the definition, requirements, existing approaches and threat model of software watermarking.

2.1 Problem Definition

The objective of software watermarking is to provide a reliable identification service to be able to claim the ownership of a software product. In a nutshell, software watermarking consists of two separate processes: i) embedding a unique signature and ii) extracting the signature for verification. Formally, the processes of software watermarking are defined as follows:

**Definition 1.** Given an original program ($P$) and a watermark ($W$), software watermarking consists of two functions; i) a watermark embedder function is $F_{\text{embed}}(P, W) = P_W$ where $P_W$ is a program with the embedded watermark $W$, and ii) a watermark extractor function, $F_{\text{extract}}$, extracts the watermark $W'$ from $P_W$ with metadata $M_e$, and verifies the extracted watermark $W'$ with

$$F_{\text{extract}}(P_W, M_e) = \begin{cases} W' & \text{if } W = W' \text{ (Valid)}, \\ -1 & \text{if } W \neq W' \text{ (Invalid)} \end{cases}$$

2.2 Requirements

As with previous work [14, 21, 23, 27, 42, 49, 67] on software watermarking, we informally define six key properties (metrics) to evaluate the effectiveness of the watermarking scheme. Note that any watermarking system exhibits a trade-off between these metrics; a high capacity (data rate) implies low stealth and resilience.

- **Resiliency:** A watermark must be robust against varying corruption attempts: watermark invalidation, tampering, addition or deletion. Moreover, even when a target software with the watermark has been altered, an ideal watermark scheme should maintain its validity or (at least) remain partially recoverable.
- **Spread:** An ideal watermark should be distributed all over a program to protect as many parts as possible. A well-distributed watermark offers probabilistically better resiliency.
- **Credibility:** A watermark should be reliably recoverable for the proof of the authorship. A false positive case (i.e., extracting a watermark from software without a watermark) or false negative case (i.e., failing to extract a watermark from software with a watermark) should be minimal.
- **Capacity:** A watermarking algorithm should be able to convey a certain amount of information (i.e., data rate) within a target program. It is desirable to quantitatively compute the maximum length of the watermark that can be encoded inside the program.
- **Efficiency:** A watermarking scheme should have a negligible impact on a target program in terms of performance and space overhead.
- **Imperceptibility:** A watermark should be stealthy (like invisible or inaudible data from video/audio files) enough not to be detected by an adversary. A program with the watermark must be indistinguishable from another without the one.

2.3 Threat Model

It is a common belief that, a determined adversary with a sufficient amount of resources will eventually be able to defeat any watermarking systems. Hence, our objective is to develop a watermarking technique that substantially thwarts every reasonable effort with feasible resources in practice, rather than building an unbreakable scheme. With this in mind, in this section, we describe a threat model with several assumptions, followed by a group of viable attacks.

**Code Signing.** A program can be digitally signed to prevent unauthorized changes [34]. Code signing involves with a cryptographic signing process using a public/private key pair that uniquely belongs to a program owner where the public key has been certified.
from the trusted third party. By signing code, an adversary’s ability to damage a watermark is severely limited, because code modification is disallowed. However, code signing does not provide traceability of the binary, which SoftMark aims to provide. Code signing is a common practice: most benign software products are signing codes, and even malware campaigns take advantage of it. Hence, it is reasonable to assume that a software copyright owner intends to leverage both code signing and watermarking to better protection, however, we assume that a target program may or may not be digitally signed.

**Attack Scenario.** We assume an adversary who may i) perform code manipulation or transformation when code signing has not been applied, ii) have the knowledge of the way how a watermark can be embedded, iii) collusively possess multiple program instances where each of which contains a different watermark, and iv) have a watermark extractor without the details of a program. We assume that the attacker does not have access to both the master binary (i.e., metadata containing function locations) and bookkeeper (or ledger) that contains useful information pertaining to a watermark (e.g., key functions used for the watermark, as described in §4.3.3), because an adversary can create a new unauthorized watermark with those information. Note that protecting the master binary and the bookkeeper from leakage (e.g., insider threat [8, 9]) is out of scope.

**Attack Types.** A robust software watermarking scheme should be able to thwart different types of attacks under our threat model. We classify such threats into three major groups as follows.

- **Unauthorized Detection** represents a risk that an attacker recognizes the presence of a watermark within a program. Hiding the presence of the watermark is of utmost importance, while developing a perfect detection-proof watermark is extremely difficult due to unexpected side channels. This attack corresponds to the property of imperceptibility.

- **Illegal Corruption** encompasses exhaustive attacks that aim to destroy a legitimate watermark by i) insertion (i.e., additive attack that attempts to implant another valid watermark), ii) deletion (i.e., subtractive attack that attempts to completely eliminate a valid watermark), iii) alteration (i.e., tampering attack that attempts to counterfeit part or full of a valid watermark), or iv) distortion (i.e., ambiguity attack that attempts to puzzle a detector by applying semantic-preserving code transformations on a target program). These attacks correspond to the properties of credibility, resiliency and spread.

- **Collusive Attack** aims to identify the location of a watermark by comparing multiple instances wherein different watermarking fingerprints have been embedded. A successful collusive attack leads to the location of or specific pattern (rule) of a watermark, without using a legitimate watermark extractor.

### 3 DEMONSTRATIVE EXAMPLE

Since SoftMark inserts a watermark via the order of functions (§4), a major threat arises from attacks leveraging semantic-preserving code transformation. This section demonstrates watermark embedding and extraction of SoftMark with an example, focusing on the difficulty of the transformation even with full accuracy.

**Target Program.** The original program in Figure 1 (a) has three functions that contain various code constructs including direct call/jump (e.g., `0x400688` and `0x400790`) and indirect call/jump (e.g., `0x400681` and `0x400893`) instructions. SoftMark would select a set of functions from all function candidates for reordering.

**Watermark with a Function Order.** Given the three functions in the example, six (≈ 3!) possible orders can represent up to two
bits as in Table 1. In principle, an individual watermark has a one-to-one mapping with a particular order of selected functions; e.g., $11_2$ can be encoded at the order of $F_1-F_2-F_3$ in Figure 1 (b).

<table>
<thead>
<tr>
<th>Function Order</th>
<th>$F_1-F_2-F_3$</th>
<th>$F_2-F_3-F_1$</th>
<th>$F_3-F_1-F_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Watermark Value</td>
<td>00_2</td>
<td>01_2</td>
<td>11_2</td>
</tr>
</tbody>
</table>

**Table 1: Function Order and Watermark Mapping.**

**Watermark Embedding.** In our scheme, inserting a watermark essentially means generating a variant of the original program with relocated functions where the order of functions representing the watermark. Such code transformation inherently involves with a vast number of updating instructions such as immediate operands. Going back to the example, direct call/jump instructions (e.g., E8 or E9 in x86) can be trivially updated by recalculating the immediate operands. However, indirect call/jump instructions require reference updates in a jump table that resides in the data region which is non-trivial. A runtime error would occur if any exercising code pointer update were failed. Moreover, successful function relocation requires a clear function boundary because it may break the original semantics otherwise.

To exemplify, the values at 0x400998 in (b) in the data section point to the call instructions at 0x4006FC and 0x400706 that have been relocated from 0x40089D and 0x4008A7. If any of those addresses has not been updated properly, the program would cause a runtime error. Similarly, the function pointers at 0x400210 in (b) that point to the function 2 and 3 must be appropriately updated according to the functions’ new addresses. This imposes a non-trivial challenge to those who attempt to compromise our watermark by relocating functions. Moreover, another challenge is to identify an accurate boundary between code pointers and raw data. In this example, the values in purple at 0x402040 in (a) and 0x400220 in (b) are scalar data (i.e., not code pointers) between two jump tables, which are indistinguishable from surrounding code pointers. To launch a successful attack, an adversary should be able to differentiate the boundary of code and data, which is undecidable. **SoftMark** takes advantage of a special compilation toolchain [35] that produces metadata for reliable static binary instrumentation (e.g., function boundary and jump table), and record unique information for a watermark when generating a mutation corresponding to the watermark. Note that the metadata produced by [35] is critical and kept secret from adversaries (Details in §7).

**Watermark Extraction.** It is straightforward to extract an embedded watermark. We can identify the order of functions with the recorded information, followed by decoding a watermark according to Table 1. Ensuring the integrity of a target binary, we discuss the case when the binary has been compromised in §4.4.

4 SOFTMARK DESIGN

This section describes the design of **SoftMark** that satisfies the requirements against various attacks when embedding and extracting a watermark.

4.1 Overview

Figure 2 depicts a workflow of **SoftMark**. First, we employ a special compiler toolchain [35] to compile a given program from the source code. During the compilation, the toolchain generates metadata (e.g., locations of functions) for reliable static binary instrumentation, required for our watermarking embedding. We call the pair of the binary and metadata master binary. Second, we analyze the binary and choose $n$ watermarking embedding candidates that can represent $k$ bits of data with different orders of the $n$ functions. Then, we generate a variant of the target program with a unique fingerprint via reordering of $n$ functions. We also record the fingerprint and its associated identifier in a ledger (accessible merely by a product owner). Third, we extract the watermark from a binary by identifying the function order. Finally, a user associated with the extracted watermark is identified by looking it up the ledger.

4.2 Benefits of Our Approach

The benefit of a static approach, including **SoftMark**, is twofold: i) inexpensive; it can be easily adopted in large-scale applications at a low cost, ii) robust; a dynamic approach relatively suffers from watermark corruption as reversing techniques advance.

**Advantages over Existing Techniques.** We aim to mitigate previous drawbacks to meet the requirements of software watermarking (§2.2). Our function-reordering-based watermark approach offers the following three advantages. First, reordering functions is a semantic preserving transformation; that is, watermark insertion does not affect the original program’s semantic because **SoftMark** does not introduce any additional code, blocks or subroutines to a target program. While the relocated functions may change cache behaviors at runtime, our assessment demonstrates that its impact on the performance overhead is negligible (§6.5.2). Second, introducing no supplementary structure gives a relatively lower chance for attackers to recognize the presence of a watermark with a statistical analysis or inference. One conceivable scenario is a collusive attack that acquires multiple instances with different watermarks, which may unveil the presence of a watermark (i.e., by identifying the locations of the same functions between the instances). Nonetheless, our watermark stays resilient against any attempt of watermark extraction (§4.4) because the mapping information between a watermark and an order is still concealed in a private ledger. Third, the number of reorderable functions can reach up to an increasingly large number of encodings (i.e., $n!$ with $n$ functions); e.g., 10 different functions can produce millions of permutations, offering a high data-rate encoding as the size of an application (and typically the number of functions) increases.

**Existing Reordering-based Approaches.** Reordering-based techniques are the closest existing approaches to **SoftMark**. However, unlike **SoftMark**, they suffer from three major limitations. First, they are perceptive; Myles et al. [42]’s approach could be easily detectable because its implementation relies on inserting a large number of GOTO statements to maintain the original control flow. Second, they are forgeable; rearranging a structure can be accomplished with a trivial effort [52, 53, 55]. Third, they are fragile; watermarks were not resilient to arbitrary modifications at the instruction level [28].

4.3 Watermark Embedding

In this section, we develop various techniques used in **SoftMark** to enhance the effectiveness of watermark embedding.
We do not use such functions in our watermark embedding because we do not consider operands (e.g., immediates) because they should be updated to obey the original flow while displacing a function.

### 4.3.2 Embedding Strategies

A watermark embedding process must support a deterministic extraction process without ambiguity. Besides, it should offer both credibility and robustness against watermark corruption attempts. We develop strategies for deterministic, credible, and robust watermark embedding and extraction.

**Unique Function Candidates.** A program may contain multiple functions that have indistinguishable (i.e., identical) binary code. We do not use such functions in our watermark embedding because our extractor cannot distinguish the locations of the functions (as well as the order between them). As an example, consider that a program consists of four functions $F_1$, $F_2$, $F_3$, and $F_4$ where the last two are indistinguishable, then a watermark with an order of “$F_3$, $F_1$, $F_4$, $F_2$” could be interpreted as “$F_4$, $F_1$, $F_3$, $F_2$”. Such multiple interpretations raise a false positive case, violating the property of credibility. It is noteworthy mentioning that the selected functions in Table 2 should be all unique. Toward uniqueness of candidate functions, we define a unique function as the one that comprises a unique combination of basic blocks where each block has a unique sequence of instruction mnemonics.

**Exclusion of Small Functions.** We empirically discover that approximately 76% of non-unique functions, on average, consists of a single basic block or even a single instruction in the programs for our evaluation (Figure 4). Note that, in such small functions, it is not rare that two functions have an identical sequence of instruction mnemonics.

**Desirable Function Set.** Once we have the list of candidates to choose from, we carefully pick functions that make static binary instrumentation challenging. When we select a candidate, we prefer a function that has a trampoline containing code pointers (i.e., indirect jumps or calls) because displacing such a function raises the bar. Specifically, an attacker needs to update a data region that embodies both code pointers and scalar data values for successful binary instrumentation. The column “ICFT (indirect Control Flow Transfer)” in Table 4 shows the number of desirable functions containing indirect branches.

**Basic Block Reordering.** We apply a basic block transposition within a function against collusive attacks. Note that this does not

---

3We do not consider operands (e.g., immediates) because they should be updated to obey the original flow while displacing a function.
a watermark extraction is straightforward and precise without vagueness because we solely use unique functions for watermark embedding.

4.4.3 Extraction from Modified Binary. If a target binary has been corrupted, the information in a ledger for the master binary does not match, raising a failure of both watermark detection and extraction. In such a case, SoftMark attempts to extract a watermark from unmodified code parts because the authenticity of a corrupted function and its location cannot be trusted. As SoftMark can insert multiple watermarks across a wide range of original code, a partial extraction of those may sufficiently reveal the fingerprint.

5 IMPLEMENTATION

This section briefly describes SoftMark implementation. Our prototype currently supports ELF executables for the x86-64 platform on Linux. We leverage the CCR [35] toolchain based on a modified LLVM (v3.9.0) and gold linker (v.2.27) to relocate functions and basic blocks. We developed our binary metadata analysis tool in Python, which takes a master binary that contains metadata for watermarking as an input. We use the ppy1ftools [7] for parsing an ELF format and the capstone [24] library as a disassembly engine.

Basic Block Pattern Search. We implemented two different techniques for seeking basic blocks as part of a function identification phase in Figure 2. By default, SoftMark discovers blocks in a ledger with a pattern using regular expressions under the assumption that a given binary has not been compromised, which allows for a quick search without a hassle. However, in case that the given binary has been modified (e.g., code transformation), it requires a deep search with a full disassembly process by matching instruction opcode and size (for recognizing the boundaries of blocks and functions). There is a coincidental case to take into account with the deep search where a consecutive functions have overlapping blocks. For example, if two functions of $F_1$ and $F_2$ share blocks like $F_1 = \{B_1, B_2, B_3\}$ and $F_2 = \{B_4, B_5, B_6\}$, it would be problematic when SoftMark mistakenly recognizes $F_1$ that comes from two blocks, say $(B_1)$ from $F_1$ and $(B_2, B_3)$ from $F_2$. Although we empirically observe that this rarely happens, we intentionally avoid such cases by re-embedding a watermark. Table 4 illustrates the comparison of embedding and extraction time between the regular-expression and disassembly-oriented implementation, whose difference is as orders of magnitude as large. Thus, it is recommended to try a deep search with full disassembly when only needed for further investigation.

6 EVALUATION

We evaluate SoftMark on a 64-bit Ubuntu 18.04 system equipped with Intel(R) Coreâ® i7-6700 3.40 GHz and 8GB RAM.

Corpus. We collect 26 binaries for SoftMark evaluation from various datasets including eight programs from SPEC CPU2006 [17], 11 samples of Binutils v2.27 from GNU Project [25], and seven utilities of our choice (e.g., putty/pscp/psftp v0.75 [58], vsftpd v2.3.4 [60], ctags v5.9.0 [19], and lighttpd v1.4.32 [38]). We have excluded applications that do not contain sufficient function candidates (i.e., at least 58 functions or above for embedding a 256-bit identifier in our experiment) such as bzip2, mcf, and specrand.

Figure 4: Ratio of small functions (i.e., containing a single basic block) of all non-unique functions. It ranges from 60% to 95% with an average of 76% (dotted line).
Table 3: Precision, recall and F1 scores of function boundary detection with IDA Pro [29]. Identifying clear boundaries from stripped binaries using a state-of-the-art reversing tool is insufficient for binary instrumentation.

<table>
<thead>
<tr>
<th>Program</th>
<th>Precision</th>
<th>Recall</th>
<th>F1 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>0.828</td>
<td>0.611</td>
<td>0.703</td>
</tr>
<tr>
<td>ar</td>
<td>0.832</td>
<td>0.414</td>
<td>0.553</td>
</tr>
<tr>
<td>gcc</td>
<td>0.804</td>
<td>0.576</td>
<td>0.671</td>
</tr>
<tr>
<td>gcc</td>
<td>0.839</td>
<td>0.653</td>
<td>0.742</td>
</tr>
<tr>
<td>gobmk</td>
<td>0.830</td>
<td>0.245</td>
<td>0.379</td>
</tr>
<tr>
<td>hammer</td>
<td>0.880</td>
<td>0.505</td>
<td>0.642</td>
</tr>
<tr>
<td>sjeng</td>
<td>0.777</td>
<td>0.626</td>
<td>0.693</td>
</tr>
<tr>
<td>test</td>
<td>0.856</td>
<td>0.693</td>
<td>0.755</td>
</tr>
<tr>
<td>sphinx3</td>
<td>0.843</td>
<td>0.594</td>
<td>0.697</td>
</tr>
<tr>
<td>addr2line</td>
<td>0.827</td>
<td>0.401</td>
<td>0.540</td>
</tr>
<tr>
<td>bfd</td>
<td>0.816</td>
<td>0.396</td>
<td>0.533</td>
</tr>
<tr>
<td>cxxfilt</td>
<td>0.836</td>
<td>0.406</td>
<td>0.547</td>
</tr>
<tr>
<td>nm-new</td>
<td>0.843</td>
<td>0.421</td>
<td>0.562</td>
</tr>
<tr>
<td>objcopy</td>
<td>0.826</td>
<td>0.464</td>
<td>0.594</td>
</tr>
<tr>
<td>objdump</td>
<td>0.818</td>
<td>0.475</td>
<td>0.601</td>
</tr>
<tr>
<td>ranlib</td>
<td>0.798</td>
<td>0.400</td>
<td>0.533</td>
</tr>
<tr>
<td>size</td>
<td>0.840</td>
<td>0.406</td>
<td>0.547</td>
</tr>
<tr>
<td>strings</td>
<td>0.803</td>
<td>0.393</td>
<td>0.527</td>
</tr>
<tr>
<td>strip-new</td>
<td>0.831</td>
<td>0.470</td>
<td>0.600</td>
</tr>
<tr>
<td>ctags</td>
<td>0.845</td>
<td>0.576</td>
<td>0.685</td>
</tr>
<tr>
<td>lighttpd</td>
<td>0.799</td>
<td>0.735</td>
<td>0.766</td>
</tr>
<tr>
<td>objdump</td>
<td>0.854</td>
<td>0.791</td>
<td>0.821</td>
</tr>
<tr>
<td>objdump</td>
<td>0.802</td>
<td>0.621</td>
<td>0.708</td>
</tr>
<tr>
<td>puttygen</td>
<td>0.791</td>
<td>0.622</td>
<td>0.697</td>
</tr>
<tr>
<td>puttygen</td>
<td>0.780</td>
<td>0.515</td>
<td>0.620</td>
</tr>
<tr>
<td>cxxfilt</td>
<td>0.789</td>
<td>0.524</td>
<td>0.630</td>
</tr>
</tbody>
</table>

6.1 Resiliency
As SoftMark operates on watermark embedding and extraction solely at a binary level, we consider possible corruption attacks focusing on machine code.

Distortion to Complicate Watermark Extractor. As described in §2.3, our SoftMark scheme may be susceptible to a distortion attack with semantic-preserving code transformation [36, 45] because such an attack impedes the proof of original program’s authenticity during a watermark extraction process. Note that while those techniques do not change the location of a function, they may challenge a watermarking extraction process by breaking the integrity of code. Our disassembly-oriented basic block search is robust to operands distortion (e.g., register reordering and assignment) but opcode distortion (e.g., instruction substitution and reordering) may lower a survival rate (See §7 in detail). Note that the aforementioned attacks can be simply prevented with code signing.

Function Relocation. As SoftMark relies on function reordering, it is susceptible to attacks that can relocate functions. Table 3 shows precision, recall, and F1 score of the function boundary detection technique implemented in the state-of-the-art disassembler, IDA Pro v7.2 [29]. The precision, recall, and F1 scores are 0.823, 0.521, and 0.629, on average, respectively. The results show that SoftMark successfully imposes significant challenges to adversaries in practice. We take ground truths from unstripped binaries with debugging information.

Other Obfuscation Techniques. We review varying obfuscation techniques offered by Sandmark [13] and others [13, 16, 31, 56, 62] to see whether they can be used to corrupt (i.e., attack) SoftMark. Since most approaches in Sandmark leverage obfuscation or optimization at a source code or Java Bytecode level to generate a valid watermark, which are not applicable to our context (i.e., binary level), we assess techniques that can be applied at a machine code level as following:

- **Code Insertion**: This attack aims to insert additional instructions (e.g., code displacement [36]), basic blocks, or functions. It will inevitably change the size of a target binary, rendering an attempt of distortion detectable. This partially thwarts our scheme by hindering function identification properly.
- **Constant Modification**: An attacker may corrupt constant values in a data region. However, it does not affect a SoftMark’s watermark, as our watermark scheme merely relies on the order of functions.
- **Basic Block Reordering**: An attacker may reorder basic blocks (as there are basic block reordering watermarking techniques) in a function. SoftMark is resilient to this type of attack since we recognize a target function with unique basic blocks that are agnostic to their orderings.
- **Branch Instruction Modification**: Modifying branch instructions may also complicate our scheme because it can alter an instruction opcode as well as a control flow change.

6.2 Spread
We evaluate how pre-selected functions are spread throughout a binary in SoftMark. Well spread functions enable a watermark to be robust against random modification of a binary; in other words, a successful attack requires a wider range of compromising code under our scheme (e.g., altering instructions or reordering functions). Table 5 shows the probability of a successful attack by the size of a watermark. Simply put, a watermark would be corrupted (i.e., failed to be extracted) when an adversary randomly chooses and alters a set of functions via distortion or reordering. Note that 433.milc and 458.sjeng do not have the results for 512 bits due to the lack of function candidates to represent those bits. Figure 6
4.3 Credibility

A watermark is embedded based on a one-to-one mapping relationship of function order information. Hence, SoftMark can precisely identify the watermark as long as all functions are unique and successfully extracted. Note that we exclude functions that may cause a false positive case as described in §4.3.2. Recognizing the functions can be complete with a basic block pattern search described in §5.

Table 4: Experimental Evaluation Dataset and Results. Pre Anal., Em., Ex., iCFT and O/H represent pre-analysis, embedding, extraction, indirect control flow transfer and a performance overhead, respectively.

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Layout</th>
<th>Functions Candidates</th>
<th>Time with a regular expression (sec)</th>
<th>Time with a disassembly (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>4,660</td>
<td>46,682</td>
<td>482</td>
<td>896</td>
</tr>
<tr>
<td>403.gcc</td>
<td>4,329</td>
<td>118,085</td>
<td>1,924</td>
<td>2,206</td>
</tr>
<tr>
<td>433.milc</td>
<td>200</td>
<td>5,350</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>3,306</td>
<td>2,720</td>
<td>34</td>
<td>105</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>1,080</td>
<td>1,400</td>
<td>24</td>
<td>72</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>338</td>
<td>5,350</td>
<td>152</td>
<td>155</td>
</tr>
<tr>
<td>cgtest</td>
<td>891</td>
<td>7,314</td>
<td>514</td>
<td>335</td>
</tr>
</tbody>
</table>

Table 5: Attack probabilities by the size of a watermark. The number of selected functions for a watermark in 400.perlbench is 58 out of 1, 660, that is, the probability of choosing the exact set of those functions is 58/1, 660 = 0.0349.

<table>
<thead>
<tr>
<th>Name</th>
<th>Attack probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>0.0127 0.0211 0.0349 0.0596</td>
</tr>
<tr>
<td>403.gcc</td>
<td>0.0049 0.0081 0.0134 0.0229</td>
</tr>
<tr>
<td>433.milc</td>
<td>0.0894 0.1489 0.2468</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>0.0085 0.0141 0.0234 0.0400</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>0.0447 0.0745 0.1234 0.2106</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>0.1591 0.2652 0.4394</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>0.0405 0.0676 0.1120 0.1911</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>0.0660 0.1101 0.1824 0.3113</td>
</tr>
</tbody>
</table>

Figure 6: Probability curve for successful attacks, which indicates that the higher number of functions or the lower capacity of a watermark, the lower probability of the attacks.

Figure 7: Probability curve for successful attacks, which indicates that the higher number of functions or the lower capacity of a watermark, the lower probability of the attacks.
6.4 Capacity
We compare our approach with the one from Davidson et al. [22] that is based on basic block reordering in terms of capacity. As shown in Table 6, the data rate of SoftMark is significantly higher than that of the Davidson’s approach (up to 15 times for 482.sphinx3). This is because our approach depends on the number of possible function candidates, in contrast, Davidson’s approach predominately relies on the maximum number of basic blocks within a function. The downside of the latter approach arises from which the largest number of basic blocks has nothing to do with the size of a program, which may not be sufficient to represent a watermark.

Table 6: Comparison of capacity (i.e., maximum number of representable bits) between SoftMark and Davidson’s approach [22] that relies on the largest block size in a function.

<table>
<thead>
<tr>
<th>Program</th>
<th>SoftMark Functions</th>
<th>SoftMark Bits</th>
<th>Davidson-Myhrvold Basic Blocks</th>
<th>Davidson-Myhrvold Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perfbench</td>
<td>1,425</td>
<td>491</td>
<td>681</td>
<td>451</td>
</tr>
<tr>
<td>403 gcc</td>
<td>3,728</td>
<td>21,326</td>
<td>534</td>
<td>4,073</td>
</tr>
<tr>
<td>433.milc</td>
<td>148</td>
<td>446</td>
<td>20</td>
<td>61</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>3,924</td>
<td>7,119</td>
<td>135</td>
<td>765</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>339</td>
<td>1,532</td>
<td>46</td>
<td>191</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>156</td>
<td>382</td>
<td>74</td>
<td>357</td>
</tr>
<tr>
<td>461.bzip2ref</td>
<td>685</td>
<td>1,708</td>
<td>160</td>
<td>945</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>210</td>
<td>909</td>
<td>20</td>
<td>61</td>
</tr>
</tbody>
</table>

We confirmed that each binary with a watermark for evaluation shown in Table 6, the data rate of SoftMark is significantly higher than that of the Davidson’s approach (up to 15 times for 482.sphinx3). This is because our approach depends on the number of possible function candidates, in contrast, Davidson’s approach predominately relies on the maximum number of basic blocks within a function. The downside of the latter approach arises from which the largest number of basic blocks has nothing to do with the size of a program, which may not be sufficient to represent a watermark. For example, 403.gcc has 2.5 times more functions than 400.perfbench, however, the maximum representable bits is rather 20% smaller. We discuss the capacities of other watermarking techniques that cannot be directly compared with SoftMark in §7.

Table 7: Differences in embedding time according to watermark values and size changes (related to Figure 7, 8). Embedding time only shows a difference of less than 1 second on the alteration of a watermark value or size.

<table>
<thead>
<tr>
<th>Name</th>
<th>Embedding Value (256 bits)</th>
<th>Embedding Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Value #1</td>
<td>Value #2</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>1.8</td>
<td>1.8</td>
</tr>
<tr>
<td>nm-new</td>
<td>10.0</td>
<td>10.4</td>
</tr>
<tr>
<td>objdump</td>
<td>21.5</td>
<td>21.8</td>
</tr>
<tr>
<td>puttygen</td>
<td>2.1</td>
<td>2.1</td>
</tr>
</tbody>
</table>

6.5 Efficiency

6.5.1 Size Overhead. The size of a watermark-inserted binary stays identical because SoftMark does not introduce additional structures such as codes, blocks or subroutines to a target program (§4.3). We confirmed that each binary with a watermark for evaluation does not increase a code size.

6.5.2 Performance Overhead. The rightmost column in Table 4 shows the performance overheads of SPEC CPU2006 binaries after embedding a watermark with SoftMark. For each binary, we measured the overall CPU user time for the completion of all internal tests by taking the average time across five runs, using both the original and its corresponding variant with a watermark. The largest overhead is reported with 456.hmmer. 1.1%, which is negligible. Interestingly, the performance of 400.perfbench, 433.milc, 445.gobmk, and 458.sjeng demonstrates slightly better than their original (master) binaries. We attribute those speedups in better caching behavior from a code region due to different code localities after function relocations, which aligns with the results from [35].
functions; medium), and objdump (988 functions; large). Empirically, we confirmed that a watermark value does not have large variations for embedding and extraction (Table 7).

Different Sizes. As shown in Table 7, embedding time is consistent on the different sizes of watermarking. However, we observe that the extraction of a watermark with a high-data rate takes longer time than that with a low rate. Interestingly, there are a few cases where extracting a 64-bit watermark is slower than 128-bit. It turns out that the functions in a 64-bit watermark contains more basic blocks. This is because our extraction performance depends on the complexity of a function for function identification. In general, embedding a larger watermark requires more functions, increasing the chance of dealing with a complicated function.

6.6 Imperceptibility

As discussed in §4, SoftMark does not add any explicit structure to a binary, which remains little information behind. Hence, inspecting a single mutation would not reveal any sign of a watermark. However, the parties in collusion who are aware of the principle of our watermark scheme (i.e., reordering-based) may learn the presence of a watermark by collecting multiple instances. Although SoftMark is equipped with several techniques to complicate function recognition (e.g., by basic block reordering within a function when generating a variant), a collusive attack can eventually thwart imperceptibility.

7 DISCUSSION AND LIMITATION

This section discusses future research and limitations of our work.

Binary Packing. Every watermark scheme on a binary code would be affected by binary packing because it involves with a widely destructive process for a code region. When a program with a watermark is packed, it is required to unpack/dump the program on memory at runtime, followed by performing watermark verification on top of the dumped code. If unpacking were failed (e.g., customize packer), SoftMark cannot reveal a watermark.

Semantic-preserving Code Transformation. ORP [45] proposes four in-place code randomization techniques without maintaining the size of a program: ① instruction substitution where an adversary replaces an instruction with another that is semantically equivalent, ② instruction reordering within a basic block by pre-computing possible orderings of given instructions, ③ register reordering with the pair registers on the stack for a function prologue (e.g., push) and epilogue (e.g., pop), and ④ register reassignment by reallocating swappable registers with pre-computing live regions in a function. As stated in §6, our scheme (even with multiple watermarks) cannot fully thwart such semantic-preserving code transformation attacks by altering opcodes (① and ②), which we leave part of our future work. An instruction displacement technique [36] demonstrates another possible code transformation but it alters a program control flow with a jmp and its size, which can be easily perceptible. Egalito [40] allows arbitrary modification at a binary level with a layout-agnostic intermediate representation, however, it only supports a position-independent executable (PIE) for rewriting a binary. Besides, SoftMark can still judge a given program that a watermark has been corrupted when it may have failed the watermark extraction.

Collision with a Function Relocation. Note that even for an attacker who can reorder functions, the probability of finding a successful collision (i.e., legitimate entry) is extremely low without a ledger. The estimated time for an accidental collision with a brute-forcing attack can be computed as the following equation:

\[
\frac{\# \text{ of all possible cases} \times \text{binary instrumentation time}}{\# \text{ of valid watermarks (collision)}} \times \frac{1}{2}
\]  

(2)

Note that we divide in half due to a 50% chance with a linear search. Assuming a computation power with Intel i7-6700 3.40 GHz and 8GB RAM for binary rewriting and a million watermarks (i.e., valid copies) available, it would take \(2.30 \times 10^{19}\) years for 458. sj eng. It is noteworthy mentioning that both SoftMark and code-signing can effectively thwart any attempt pertaining to code transformation.
Constraints on Function Relocation with CCR. To avoid introducing new instructions for binary instrumentation, CCR inherently restricts the positions for relocating functions when the size of a reference (e.g., operand for a relative jump or call) is not large enough (e.g., one or two bytes). We obey the same constraint with CCR, however, the rate of such limited relocations is small (around 1%), which rarely affects the capacity of SoftMark. For example, in gcc in our dataset has 51 out of 4,329 functions (1.12%) were constrained by this limitation.

Capacity of Other Existing Techniques. Along with §6.4, we discuss the capacity of other software watermarking techniques that cannot be directly compared to SoftMark. Sha et al. [52] leverages an equation’s operand coefficient to encode a watermark in Java programs, whose data rate is comparable to SoftMark because it uses a permutation of the coefficient. A branched-based technique [26, 43] relies on the number of branch instructions for embedding a watermark. Although it could hold a higher data rate even for a small program that contains many branches, the possible encoding capacity overall may be fluctuating. Meanwhile, an obfuscation-based approach [68] defines a hard-coded limit of 1,000 different instruction groups (i.e., 1,000 bits), which is difficult to be expanded. Several other works [18, 20, 21, 57] demonstrate a scheme that allows one to embed a unlimited watermark in size by adding additional data or method into a binary. However, such approaches are highly susceptible to be perceptible and thus easily eliminated. A graph-based approach [12, 14, 46, 59, 65, 69] generates a topological structure at runtime when a certain input is given. While they have unlimited capacity since they explicitly add code segment for the watermark, they are trivially detectable due to the added code and data.

8 RELATED WORK
A variety of software watermarking schemes have been proposed for the last two decades [23, 28, 70]. Software watermark technique can be classified as either static [4, 22, 30, 41, 51, 53, 55] or dynamic [14, 18] according to the way of extraction, that is, a static watermarking does not need to run a program whereas dynamic watermarking does because a watermark can be extracted at runtime (i.e., the execution state of the program). Note that static watermarking is much more common because it is relatively handy. In this section, we outline a major approaches for software watermarking techniques and CCR [35], a compiler-rewriter model for our static binary instrumentation.

Reordering-based Approach. Diversifying code is one of promising techniques for securing and protecting software since early days. The idea of early patents [30, 51] places an identifier into a pre-determined (and random) location of code or data. Similarly, Davidson et al. [22] introduces a means of inserting a signature by relocating a group of pre-selected basic blocks. Shirali-Shahreza et al. [55] suggest an equation reordering technique that swaps the safe operands of mathematical equation in source code, and later FDOS [53] introduces a scheme of function dependency-oriented sequencing on top of reordering equations. Although the basic idea of “reordering” aligns with our SoftMark, the above approaches are susceptible for i) revealing (resiliency) as it merely relies on localized piece of code; ii) being removed as a watermark is not widely spread (i.e., poor part protection), and iii) insufficient data rate as it depends on the largest component (e.g., number of functions or operands) that limits encoding bits, and iv) reliable binary instrumentation when inserting a watermark at a binary level lacks [22].

Graph-based Approach. Another line of static watermarking is based on a graph theory [14, 32, 48, 49]. Qu et al. [48] apply a graph coloring (GC) problem to a register allocation of variables, which inserts a watermark by adding edges in a given graph of \( G(V, E) \). Later, Jiang et al. [32] presents a software watermarking scheme based on public-key cryptograph with GC. However, graph coloring has no efficient algorithm (known NP complete problem). Collberg et al. [14] proposed a dynamic watermarking technique (dubbed CT) that is stored in the execution state of a program (e.g., through a graph structure on the heap).

Obfuscation-based Approach. Balachandran et al. [4] suggest an obfuscation algorithm that interfaces blocks across functions with anti-disassembly techniques for concealing them. Monden et al. [41] demonstrates the insertion of watermark into dummy methods and opaque predicates in Java programs. Myles et al. [42] carefully analyze the effectiveness between the Davidson’s reordering-based approach [22] and Monden’s obfuscation-based approach [41] with actual implementations using the Sandmark tool [13]. Lu et al. [39] propose an obfuscation-based steganography technique by leveraging ROP gadgets to embed certain information that can be extracted at runtime by running the ROP gadgets. While steganography has a slightly different purpose from watermarking, we believe it can also be used to implement a watermark.

Other Approaches. A spread-spectrum watermarking scheme [20, 57] has been suggested from the signal detection model in multimedia watermarking, extracting a vector from the properties of a running program (e.g., call graph depth). Preda et al. [21] presents a formal framework for modeling a software watermarking technique at a semantic level by viewing attackers as abstract interpreters. Cousot et al. [18] introduces a dynamic watermark scheme that leverages abstract interpretation to insert a watermark into values that are assigned to local variables at runtime. Nagra et al. [44] suggest a precise taxonomy in the area of software watermarking.

Compiler-assisted Code Randomization. Relocating functions from a stripped binary is, in general, non-trivial because of precise disassembly [2], binary function recognition [1, 3, 6, 47, 54, 61], and varying optimizations at compilation. To this end, we adopt a compiler-rewriter cooperation approach [35] that allows for robust and fast code transformation. Simply put, it stores a minimal set of supplementary information (including a layout, basic block, and fixup or reference that must be adjusted after function displacement) into a master binary as metadata, enabling us to carry out static binary instrumentation without recompilation [35, 50] on demand. The master executable is maintained along with watermarking information by a program owner where those who purchase the software possess a mutant (i.e., reordered version) with a watermark alone.

9 CONCLUSION
In this paper, we propose a function reordering-based software watermarking technique, SoftMark. It embeds a watermark, mapping
We thank the anonymous referees and our shepherd Sang Kil Cha for their constructive feedback. This work was supported by Institute for Information & communication Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2019-0-01343, Regional strategic industry convergence security core talent training business), NSF under awards 1916499, 1908021, and 1850392. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

REFERENCES


