2010 International Conference on Advances in Communication, Network, and Computing

Zero-knowledge Software Watermarking for C Programs

Sukriti Bhattacharya
Dipartimento di Informatica
Universita' Ca' Foscari di Venezia
Via Torino 155, 30170 Venezia, Italy
sukriti@dsi.unive.it

Agostino Cortesi
Dipartimento di Informatica
Universita' Ca' Foscari di Venezia
Via Torino 155, 30170 Venezia, Italy
cortesi@unive.it

***Abstract*—This paper proposes a novel method for watermarking C source code by exploiting the programming language features.The key idea of our watermarking scheme is a semantics-preserving program transformation, based on a hidden permutation of local identifiers, followed by another hidden permutation of the functions defined in the source code. This last permutation allows to encrypt the prove of ownership, in the framework of interactive zero-knowledge proof system. The proposed watermarking scheme is invisible to compilers and does not reveal any information about the watermark, its nature and its location into the program, since the zero knowledge proof is independent of the encoding and of the embedding. Finally, we introduce a third party Trusted Time-Stamp Service into the system to prevent invertibility/ambiguity attacks.**

***Keywords*-Software Watermarking, Semantics-preserving program transformation, Zero-knowledge proof systems.**

I. Introduction

With the increasing amount of program source code (most of the time in the form of bytecode) which is distributed in the web, software ownership protection and detection is becoming an issue. In particular, with multiple distributions of code, in order to prevent the risk of running fake programs, it is important to provide authentication proofs that do not overload the packages and that are easy to check. This is the aim of the so called Software Watermarking Techniques. Software watermarking embeds hidden information about ownership and integrity of the program into the code itself which can be retrieved and checked automatically on demand. In general, it is not possible to devise watermarks that are immune to all conceivable attacks; it is generally agreed that a sufficiently determined attacker will eventually be able to defeat any watermark. In our vision, watermarking is a method that does not aim to stop piracy copying, but to check the ownership of the software. Therefore in our approach watermarking is seen as an alternative to encryption as a way to support software authentication rather a tool for copyright protection.

We can classify the most relevant existing software watermarking techniques as, Graph-based software watermarking [14][3][4], register-based software watermarking [8][9], thread-based software watermarking [10], obfuscation-based software watermarking [5][11], branch-based software watermarking [6][7], program Slicing based software watermarking [19] and abstract Interpretation based software watermarking [13].

We propose a public key software watermarking (asymmetric watermarking) scheme which is similar in sprit to zero-knowledge proofs [1][12]. The main idea is to prove the presence of a watermark without revealing the exact nature of the mark. The embedding algorithm inserts a watermark W into the source code of the program P using the private key of the owner. In the verification process we established a protocol V that has access to the watermarked program P_W and to the corresponding public key. V proves the presence of watermark W in P_W without revealing the exact location of the watermark nor the nature of the watermark specified by private key. The central idea of our watermarking scheme can be defined by the following steps,

- The watermark embedding process is based on a semantics-preserving program transformation using a permutation on the set of local identifiers.
- The watermark detection and verification process is based on a permutation that produces a scrambled version of the watermarked program by reordering the functions defined in the program. This permutation is introduced to prove the ownership in a interactive zero-knowledge proof system framework.
- This watermarking scheme can be combined with a *time-stamp* mechanism that makes it robust against invertibility or ambiguity attacks.

The paper is structured as follows. In section 2, we describe the terminology required for a formal definition of zero-knowledge watermark detection. A new generic watermarking technique for C source code is proposed in section 3, where we discuss watermark generation, embedding and detection process in zero-knowledge authentication proof system. We illustrate our watermarking scheme by a suitable example in section 4. In section 5, we propose a way to handle the so the called *ambiguity* attacks by means of

978-0-7695-4209-6/10 $26.00 © 2010 IEEE
DOI 10.1109/CNC.2010.72

time-stamping to make our scheme robust. In section 6, we conclude by discussing the main advantages of our scheme.

II. Terminology and definitions

In this section, we present the definitions of the primitives which are required for a formal definition of zero-knowledge watermark detection [1][12] and that will be used in section 4 in order to properly explain our watermarking technique.

A. Semantics-preserving program transformation

Let dom(P) be the set of input sequences accepted by a program P, out(P, I) be the output of P on input I. Let $\mathbb{T}$ be the set of transformations from programs to programs. An input output semantics-preserving transformation T $\in$ $\mathbb{T}$ satisfies the properties, dom(P) = dom(T(P)) and $\forall$ I $\in$ dom(P): out(P, I) = out(T(P), I).

B. Software watermarking schemes

Let $\mathbb{P}$ is the set of programs to be watermarked. Software watermarking schemes can be defined by a tuples, S = $(G_{key}(), G_W(), E(), D())$, where

- $G_{key}()$ is a polynomial-time algorithm. On input of the security parameters, it generates the keys (K_{emb}, K_{det}), required for watermark embedding and detection.
- $G_W()$ is a polynomial-time algorithm. On input of the security parameters, it generates the watermark *W*.
- On input of a program $P \in \mathbb{P}$, a watermark *W* to be embedded and the embedding key K_{emb} the the polynomial time embedding algorithm $E(P, W, K_{emb})$ outputs the watermarked program P_W.
- On input of a (possibly modified) watermarked program P_W, the watermark *W*, the original program P and the detection key K_{det}, the detection algorithm $D(P_W, P, W, K_{det})$ outputs a boolean value, 1 for the presence and 0 for the absence of the watermark.

C. Commitment schemes

A commitment scheme, between the involved parties, a committer and a receiver enables one to fix a message such that it cannot be changed after committing, while the committed value is kept secret from anyone else.The security requirements are the binding (committing) and hiding (secrecy) properties. Informally, the first one requires that a dishonest committer cannot open a commitment in two different ways, and the second one requires that the commitment does not reveal any information about its content to the receiver.

D. Interactive proof systems

An interactive proof system is a two-party challenge-response protocol between two interactive algorithms, a *Prover* and a *Verifier*. The task of *Prover* is to prove a statement to *Verifier*, which is represented by the common input *x*. The fundamental properties of an interactive proof system (*Prover*, *Verifier*) are soundness, i.e. a cheating prover $P^{\#}$ cannot prove a wrong statement to an honest verifier V^* and completeness, i.e. A correct prover P^* can prove all correct statements (assertions) to a correct verifier V^*.

E. Zero-knowledge software watermark detection

A proof system is said to be zero-knowledge, if the system reveals no knowledge to the verifier, except the fact that the assertion is valid. Thus, zero-knowledge considers only honest provers whereas the verifier is in general considered to be an adversary, who wants to extract knowledge from the prover.
The general idea behind zero-knowledge watermark detection is as follows: Given a fixed watermark *W*, the program *P*, secret watermarking key K_{emb} and watermarked program P_W (possibly modified), the *Prover* can convince the *Verifier* that the watermark is detectable in P_W relative to *P* under the key K_{emb} without revealing any additional information about the watermark.

III. A new watermarking scheme for C programs

The watermarking scheme for C source code, we introduce now, exploits explicitly programming language features. Let $P \in \mathbb{P}$ be the program to be watermarked. Let $Id(P) = InId(P) \cup LocId(P) \cup OutId(P)$ be the identifiers of program *P*, where *InId(P)* are the identifiers defined in other functions imported by *P* (e.g. *extern* variables),*OutId(P)* are the identifiers that are exported (global variables) to other programs interacting with *P*, and *LocId(P)* are the remaining local identifiers. The central idea of our watermarking scheme can be defined by the following two steps

- The watermark embedding process is based on a semantics-preserving program transformation using a permutation π on *LocId(P)*.
- The watermark detection and verification process is based on a permutation Γ, produces a scrambled version of the watermarked program by reordering the functions defined in *P*. This permutation is introduced to prove the ownership in a interactive zero knowledge proof system framework.

A. Watermark generation

Let the owner of the program *P* possess a secured key $\Re$; the key should be long enough to thwart brute force guessing attacks. A cryptographic pseudo random sequence generator [2] *G* is seeded with key $\Re$ and the concatenated identifiers, generating a sequence of numbers. A permutation π is chosen from the permutation group S_n on n elements (assuming $|LocId(P)| = n$) on the basis of the output generated by *G* [15][16]. The permutation π is considered as a security parameter for the watermark embedding algorithm and $\pi(LocId(P))$ will be the generated watermark. Here

we adopt the function unrank following the Myrvold and Ruskey's linear permutation unranking algorithm [18] to generate the permutation π.

$\mathbf{genW(LocId(P)}, \Re)$

1: r= $G(\Re, Id_1||Id_2||...||Id_n)$ mod (n!)
2: $LocId(P) = \{Id_1, Id_2, ..., Id_n\}$
3: **for** i=1 to n **do**
4: $\pi_i = Id_i$
5: **end for**
6: $Unrank(\pi, n, r)$
7: **return**$(\pi(LocId(P)))$

Unrank(π, n, r)

1: **if** $n > 0$ **then**
2: swap ($\pi[n\text{-}1]$, $\pi[r \bmod n]$)
3: *Unrank*$(n-1, \lfloor r/n \rfloor, \pi)$
4: **end if**

B. Watermark embedding

The watermark embedding function is a semantics-preserving program transformer which just substitutes each identifier $LocId(P)$ by the corresponding one in $\pi(LocId(P))$. For the sake of simplicity we denote $LocId(P)$ as Id and $\pi(LocId(P))$ as Id_π. $E_\Re$: $P_{Id} \rightarrow P_{Id_\pi}$. Where $E_\Re$ is the watermark embedding algorithm, $\Re$ the secure watermark embedding key, P_{Id} is the program to be watermarked and P_{Id_π} is the watermarked program. So far we used traditional symmetric key watermarking algorithm to generate the watermarked program P_{Id_π}. It could be a very tedious job for the attacker to select the actual permutation from S_n in worst case scenario, since the number of identifiers(n) are supposed to be huge in a software which increases the number of permutations (n!) radically.

C. Watermark detection and verification

Assume that software owner (*Prover*) Alice inserted her watermark *W* into the program by using watermarking algorithm described in section 3.2, yielding the watermarked program P_{Id_π}. The software user (*Verifier*) Bob who is expecting the proof of authentication from Alice. To do so, both Alice and Bob participate to a challenge-response protocols in a interactive zero knowledge proof system framework as follows, Let $F(P) = \{f_1, f_2, ..., f_m\}$ be the set of functions defined in program *P*. Alice now produces a scrambled version of her software by another permutation Γ. The central idea of this scheme (scrambling) is to alter the sequence in which the functions in $F(P)$ appear in *P*, by choosing a hidden permutation Γ from the permutation group S_m. We denote the scrambled program obtained in this way by P_{F_Γ}. Alice sets up a public directory where she publishes P_{Id_π} along with P_{F_Γ}. We assume that the software consists of a large number of functions so that the size of the permutation group S_m, is large enough to make the permutation difficult to guess by brute force. Notice that P_{F_Γ} is useless with respect to program execution, because altering the sequence of order of functions in P_{Id_π} might yield to compile-time errors in C. By using the following protocol, Alice proves Bob that she actually knows the secret Γ and that her watermark is present into the program, by revealing no knowledge to Bob about the watermark generation, nor the embedding, and nor the exact location of watermark in *P*. The protocol might be better understood by looking at Fig. 1.

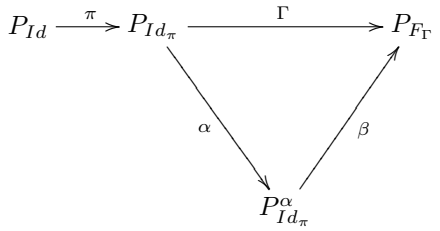

Fig 1: Watermark verification

D. Zero-knowledge Verification Protocol

- Alice generates the permutations α and β, and $\Gamma = \alpha o \beta$. She also computes $P^{\alpha}_{Id_\pi} = \alpha(P_{Id_\pi})$. Alice generates an ownership ticket (OT) with two commitments (for α and β) and a hash $H(P^{\alpha}_{Id_\pi})$. OT=$\langle C_1(\alpha), C_2(\beta), H(P^{\alpha}_{Id_\pi})\rangle$
  Alice sends the signed OT to Bob.
- Bob flips a coin and ask Alice either
  - to open commitment C_1, or
  - to open commitment C_2
- Alice responds by opening either C_1 or C_2, based on Bob's request.
- Now the following two cases can arise on Bob's side:
  - Case 1 (Alice opens commitment C_1): Bob computes $\alpha(P_{Id_\pi})$ from the knowledge of α contained in commitment C_1. Bob computes $H(\alpha(P_{Id_\pi}))$ and checks whether $H(P^{\alpha}_{Id_\pi}) = H(\alpha(P_{Id_\pi}))$.
  - Case 2 (Alice opens commitment C_2): Bob computes $\beta^{-1}(P_{F_\Gamma})$ from the knowledge of β contained in commitment C_2. Bob computes $H(\beta^{-1}(P_{F_\Gamma}))$ and checks whether $H(P^{\alpha}_{Id_\pi}) = H(\beta^{-1}(P_{F_\Gamma}))$.
- Alice and Bob perform these steps repeated number of times (*k*). If all tests pass, Bob is convinced by Alice that the watermark is present in P_{Id_π}, and that Alice is the owner of P_{Id}.

Theorem 1: The *Zero-knowledge Verification Protocol* is complete.

Proof: During the protocol, no information about Γ is leaked. If an attacker can not determine Γ, then the value of α reveals nothing about the value of β, and vice versa, since S_m is a group. For every possible secret Γ and every possible revealed permutation α, there exists one and only one β such

that $\Gamma = \alpha o \beta$, and analogously this is true for a revealed β. Therefore, the *Zero-knowledge Verification Protocol* is described above complete.

Theorem 2: The *Zero-knowledge Verification Protocol* is weakly sound: after *k* iterations of the zero-knowledge protocol a cheating prover has a success probability $1-\sum_{i=1}^{k}(\frac{1}{2^i})$.

Proof: Suppose Bob wants his own watermark to appear in the program P_{Id}. Although he cannot change P_{Id}, he is free to add his own watermark W to P_{Id} and to scramble the program with his own hidden permutation Γ chosen from S_m. He pretends that his scrambled program is the true scrambling of P_{Id} induced by his own Γ. He may construct the ownership ticket by using either permutation α or β, by fooling a verifier in one of the two cases. Therefore, the probability of his success in each round is $\frac{1}{2}$. So after *k* iterations of the zero-knowledge protocol above, we get the resulting probability equals to $((((1-\frac{1}{2})-\frac{1}{2^2}))...-\frac{1}{2^k}) = 1-\sum_{i=1}^{k}(\frac{1}{2^i})$.

IV. Example

Let us illustrate our watermarking scheme by a simple program *product.c* which performs multiplication operation by repeated addition method. *LocId(P)*= {a, b, sum, product, mul, x, y, prod, add, p, q, i, k, print_prod} and *F(P)*={mul(), add(), print_prod()}. The permutation group S_{14} consists of (14)!=87178291200 permutations and S_3 consists of (3)!= 6 permutations. Now suppose statement 1 of algorithm genW() in section 3.1 generates a hypothetical value r which is used to generate the secret permutation $\pi_r \in S_{14}$. Let $\pi_r(LocId(P))$ ={product, prod, i, b, q, sum, a, mul, y, k, add, print_prod, x, p}. The next step is to perform the second hidden permutation Γ on the watermarked program P_{π_r} to generate its scrambled public version P_{F_Γ} by reordering the $F(P)$. $P_{F_\Gamma} \in S_3$.

```
int add(int p, int q){
 int i,k=0;
 for(i=0;i<q;i++)
   k=k+p;
  return(k);}

void mul(int x, int y){
 int prod;
 prod= add(x,y);
 print_prod(prod);}

void print_prod(int product){
printf("%d", product);}

void main void(){
 int a, b, sum;
 scanf("%d",&a);
 scanf("%d",&b);
 mul(a,b);}
```

Fig.2a: Original Program

```
int y(int k, int add){
 int print_prod,x=0;
 for(print_prod=0;print_prod<add;
     print_prod++)
  x=x+k;
 return(x);}


void q(int sum, int a){
 int mul;
 mul= y(sum,a);
  p(mul);}


void p(int b){
printf("%d", b);}


void main (void){
 int product, prod, i, b;
 scanf("%d",&product);
 scanf("%d",&prod);
 q(product,prod);}
```

Fig.2b: Watermarked Program

```
void q(int sum, int a){
 int mul;
 mul= y(sum,a);
  p(mul);}


int y(int k, int add){
 int print_prod,x=0;
 for(print_prod=0;
 print_prod<add;
 print_prod++) {
  x=x+k;
 return(x);}


void p(int b){
printf("%d", b);}


void main (void){
 int product, prod, i, b;
 scanf("%d",&product);
 scanf("%d",&prod);
 q(product,prod);}
```

Fig.2c: Scrambled Program

The original program *P* and corresponding watermarked program P_{π_r} are shown in Fig 2a and Fig. 2b, respectively. The scrambled program P_{F_Γ} is shown in Fig. 2c. Notice that P_{F_Γ} (Fig.2c) generates compile time errors during compilation, The computational complexity of watermark generation algorithm is linear. The watermark generation algorithm produces a permutation selected uniformly at random from amongst all permutations in S_n. Let $r_{n-1}, r_{n-2}, ..., r_1, r_0$ be the sequence of random elements where $0 \leq r_i \leq i$. Since there are exactly $n \times (n-1) \times (n-2) \times ...2 \times 1 = n!$ each difference sequence must produce different permutation. Thus we should be able to unrank if we can take an integer r in the range $[0..n!-1]$ and turn it into a unique sequence of values $r_{n-1}, r_{n-2}, ..., r_1, r_0$ where $0 \leq r_i \leq i$. According to the *Unrank* procedure this can be done in $O(n)$ operations. The computational complexity of the watermark embedding operation is also linear. If the original program has m statements and it contains n local variables, then the semantic preserving program transformation performs $O(mn)$ operations to generate the watermarked program.

V. Security considerations

In zero-knowledge watermark detection systems, *invertibility* or *ambiguity* attacks are very well known attacks where an adversary can create an ambiguous situation by deriving a forged watermark from a published work, and commits the forged watermark. Suppose Alice and Mallory use the same watermarking technique proposed here to watermark their softwares. And suppose Mallory wants his own watermark to appear in the program (P_{Id_π}) watermarked by Alice. Although he can't change P_{Id_π}, but he is free to add his own watermark $\pi(LocId(P_{Id_\pi}))$ to P_{Id_π} and scramble watermarked program P'_{Id_π} by his own hidden permutation Γ, yielding $\Gamma(P'_{Id_\pi})$. And he pretends the owner of the

program *P*. We can prevent such ambiguities by involving a third party *Trusted Time-Stamping Service*(TTSS) [17]. Before sending the signed OT to the *Verifier* (as described in section 3.4), the *Prover* must send it to a TTSS. The TTSS records the date and time the document was received and retains a copy of OT (signed by the *Prover*) for safe-keeping. The TTSS appends a signed *time-stamp T* for the submitter (i.e. the *Prover*) using his symmetric private key K_{TTSS} and send it back to the submitter (i.e. the *Prover*). So in our example the modified OT from Alice will look like, OT=$\langle C_1(\alpha), C_2(\beta), H(P^{\alpha}_{Id_{\pi}}), (T)_{K_{TTSS}} \rangle$. Then in the *time-stamp* verification phase, both the *Verifier* and the *Prover* have to come to TTSS and on the basis of their *time-stamp* values TTSS will solve the ambiguity about by revealing the actual owner of the software.

VI. Conclusions

The proposed watermarking scheme is invisible and does not reveals any information about the watermark and its location into the program, since the zero-knowledge proof is independent of the encoding and embedding, it is free from *collusive* attacks, where the attacker may make an attempt to gather some information about the watermark. The scheme can be combined with a *time-stamp* technique to handle *invertibility/ambiguity* attacks. This approach can be easily extended also to other programming languages by applying suitable permutation-based program transformations that exploit the particular programming languages' features.

Acknowledgment

Work partially supported by Italian MIUR COFIN 07 project "SOFT" and by RAS project TESLA.

References

[1] A. Adelsbach, S. Katzenbeisser, and A.R. Sadeghi.: Watermark detection with zero-knowledge disclosure. In: Multimedia Systems, vol. 9, Springer, Berlin, Germany, 2003, pp. 266-278.

[2] B. Schneier. Applied Cryptography. John Wiley and Sons, Inc., 1996.

[3] C. Collberg, A. Huntwork, E. Carter, and G. Townsend.: Graph theoretic software watermarks: Implementation, analysis, and attacks. In: Proceedings of 6th Information Hiding Workshop, LNCS, vol. 3200, 2004, pp. 192-207.

[4] C. Collberg and C. Thomborson.: Software watermarking: Models and dynamic embeddings. In: Proceedings of Principles of Programming Languages 1999, POPL'99, 1999, pp. 311-324.

[5] C. S. Collberg and C. Thomborson. :Watermarking,tamper-proofing, and obfuscation-tools for software protection. In: IEEE Transactions on Software Engineering,vol. 28, no. 8, August 2002, pp. 735-746.

[6] C. Collberg, E. Carter, S. Debray, A. Huntwork, C. Linn, and M. Stepp.: Dynamic path-based software watermarking. In: Proceedings of Conference on Programming Language Design and Implementation, vol. 39, June 2004, pp. 107-118.

[7] G. Myles and H. Jin.: Self-validating branch-based software watermarking. In: Proceedings of 7th Information Hiding Workshop, LNCS, vol. 3727, 2005, pp. 342-356.

[8] G. Qu and M. Potkonjak.: Analysis of watermarking techniques for graph coloring problem. In: Proceedings of International Conference on Computer Aided Design, 1998, pp. 190-193.

[9] G. Myles and C. Collberg.: Software watermarking through register allocation: Implementation, analysis, and attacks. In: Proceedings of International Conference on Information Security and Cryptology, LNCS, vol. 2971, 2003, pp. 274-293.

[10] J. Nagra and C. Thomborson.:Threading software watermarks. In: Proceedings of 6th Information Hiding Workshop, LNCS, vol. 3200, 2004, pp. 208-223.

[11] M. Sosonkin, G. Naumovich, and N. Memon. Obfuscation of design intent in object-oriented applications. In Proceedings of 3rd ACM workshop on Digital Rights Management, 2003, pp. 142-153.

[12] O. Goldreich, J. Oren, "Definitions and Properties of Zero-Knowledge Proof Systems", Journal of Cryptology, 1994, 7(1), pp. 1-32.

[13] P. Cousot, R. Cousot.: An abstract interpretation-based framework for software watermarking. In: Conference Record of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Venice, Italy, 2004, pp, 173-185.

[14] R. Venkatesan, V. Vazirani, and S. Sinha.: A graph theoretic approach to software watermarking. In: Proceedings of 4th Information Hiding Workshop, LNCS, vol. 2137, 2001, pp. 157-168.

[15] S. Bhattacharya, A. Cortesi.: A Generic Distortion Free Watermarking Technique for Relational Databases". In:ICISS 2009, LNCS 5905, pp. 252-264.

[16] S. Bhattacharya, A. Cortesi. :A Distortion Free Watermark Framework for Relational Databases". In: ICSOFT (2) 2009, pp. 229-234.

[17] Stuart Haber and W.Scott Stornetta.: How to Time-Stamp a Digital Document. In: Journal of Cryptology,3(2) 1991, pp. 99-111.

[18] W. Myrvold and F. Ruskey.: Ranking and unranking permutations in linear time. Accepted by Information Processing Letters, Oct., 2000.

[19] X. Zhang and R. Gupta.: Hiding Program Slices for Software Security. In: Proceedings of First Annual IEEE/ACM International Symposium on Code Generation and Optimization, San Francisco, CA, March 2003, pp.325-336.