Fingerprinting de Software e Aplicações à Metrologia Legal

Lucila Maria Souza Bento[1,2], Rafael de Oliveira Costa[1,2], Davidson Rodrigo Boccardo[2], Raphael Carlos Santos Machado[2], Vinícius Gusmão Pereira de Sá[1], Jayme Luiz Szwarcfiter[1,2,3]

[1] Instituto de Matemática, Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil
lucilabento@ppgi.ufrj.br, rafaelcosta@ppgi.ufrj.br, vigusmao@dcc.ufrj.br
[2] Instituto Nacional de Metrologia, Qualidade e Tecnologia, Rio de Janeiro, Brasil
drboccardo@inmetro.gov.br, rcmachado@inmetro.gov.br
[3] COPPE-Sistemas, Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil
jayme@nce.ufrj.br

Abstract* — *O termo fingerprinting refere-se ao ato de embarcar uma informação num objeto com o objetivo de torná-lo posteriormente rastreável. Neste trabalho, propomos uma aplicação de fingerprinting à Metrologia Legal. Consideramos o modelo de validação de instrumentos de medição que envolve uma etapa de análise de software, e propomos o uso de uma técnica de fingerprinting na construção de um protocolo de segurança que permite identificar responsáveis por eventuais "vazamentos de código". A técnica utilizada emprega fingerprinting baseado em grafos, que são estruturas combinatórias naturalmente associadas ao fluxo de execução de um programa.

Keywords* — *Metrologia Legal, fingerprint, validação de software

I. Introdução

Instrumentos de medição envolvidos em relações comerciais estão sujeitos ao que chamamos de *controle metrológico*, por meio do qual uma agência pública – a Autoridade Metrológica Legal – avalia a confiabilidade do funcionamento do medidor. O processo através do qual um medidor é validado por uma Autoridade Metrológica Legal é conhecido como *aprovação de modelo*, e compreende a avaliação do software legalmente relevante, isto é, de todos os módulos de software envolvidos no processo de captura, processamento e exteriorização dos resultados da medição para o usuário final. A fim de assegurar o correto funcionamento do medidor, a aprovação de modelo envolve, frequentemente, não apenas a abertura do código-fonte do software legalmente relevante para a inspeção de detalhes de implementação, como também a verificação da topologia do hardware utilizado.

Um caminho possível para se agilizar a análise de código é a terceirização da avaliação do software embarcado em um instrumento de medição por laboratórios acreditados pela própria Autoridade Metrológica. Embora possa ser interessante para a Autoridade Metrológica, a terceirização da avaliação pode contudo dificultar o rastreamento e a penalização dos responsáveis por quaisquer "vazamentos de código", isto é, por cópias ou apropriações indébitas do todo ou de partes do software em questão.

Diante desse cenário, propomos um protocolo que, utilizando técnicas de *fingerprinting* baseadas em grafos, torna possível a identificação unívoca de terceiros envolvidos na avaliação dos módulos de software dos instrumentos de medição delegados pela Autoridade Metrológica, permitindo por conseguinte o rastreamento dos responsáveis por eventuais vazamentos.

II. Protocolo Proposto

Apresentamos um protocolo de segurança no qual a Autoridade Metrológica assume o papel de uma terceira parte confiável (*trusted third party* – TTP) responsável por identificar unicamente cada módulo de software distribuído para análise. A identificação de tais módulos é feita com o uso de *fingerprints*, informações que são codificadas e embarcadas no software com a propriedade de não serem facilmente removíveis. Um *fingerprint* é inserido em um programa por

um algoritmo de inserção (*embedder*) e pode ser dele recuperado por meio de um algoritmo reconhecedor (*recognizer*).

Formalmente, um esquema de *fingerprinting* é uma função $f : P \times W \times K \rightarrow P$, cuja entrada é constituída de um código de software $p \in P$, uma informação $w \in W$ e um parâmetro secreto $k \in K$, e cuja saída é um código alterado $q \in P$ tal que:

i) q possui as mesmas características funcionais de p;
ii) a partir de quase todo código q' "suficientemente próximo" de q, é possível recuperar a informação w, uma vez que se conheça k.

É importante mencionar que a função f não precisa ser secreta, isto é, qualquer indivíduo pode ser capaz de gerar q a partir de p, w e k. Por outro lado, a segunda condição acima nos diz que, de posse do segredo k, é possível recuperar a informação w a partir do código com *fingerprint* mesmo que este tenha sofrido uma modificação de amplitude limitada. Isto significa que é difícil, para um adversário malicioso, remover ou burlar a identificação provida pelo *fingerprint* que se encontre embarcado no software.

Por meio do uso de *fingerprints*, é possível identificar unicamente módulos de software que sejam distribuídos para avaliação por terceiros. Considere o cenário onde um desenvolvedor apresenta o código p para ser avaliado pela Autoridade Metrológica, e a Autoridade Metrológica deseja demandar a avaliação deste código a uma terceira instituição. Propomos o seguinte protocolo:

Protocolo:
(1) A Autoridade Metrológica gera um resumo criptográfico, ou *hash*,do conjunto do código a ser avaliado acrescido de informações que identifiquem o avaliador.
(2) A Autoridade Metrológica assina o *hash*. O conjunto hash assinado + identificação do avaliador compõem a informação w a ser embarcada.
(3) A Autoridade Metrológica codifica w na forma de um grafo apropriado, embarcando-o no programa original p (em uma região secreta k).

É importante compreender a motivação de cada uma das etapas. A etapa (1) é necessária para que a informação w a ser embarcada no código p seja única, dependente tanto do próprio código a ser avaliado como do avaliador que o venha a receber. Informações complementares podem ser acrescentadas, como a data de inserção do *fingerprint*, por exemplo. A etapa (2) garante que apenas a Autoridade Metrológica terá a capacidade de embarcar informação válida em um código; de outro modo, qualquer indivíduo seria capaz de inserir em um código a informação de que esse código encontra-se em posse de um avaliador (imputando-lhe culpa por vazamentos que venham a ser praticados), mesmo que esse avaliador jamais o tenha recebido. A etapa (3) garante que somente o avaliador designado possui aquela variante de código com determinado *fingerprint*, de forma que, após a entrega feita na etapa (4), eventuais vazamentos provenientes daquela variante poderão ter sua origem apontada pelo *fingerprint* que dela se recupere, indicando dessa forma a culpabilidade daquele avaliador.

III. Fingerprints Baseados em Grafos

O algoritmo pioneiro de *fingerprinting* baseado em grafos foi formulado por Davidson e Myrhvold [1]. Seu trabalho inspirou a apresentação, por Venkatesan, Vazirani e Sinha [2], do primeiro algoritmo de *fingerprinting* no qual uma chave de identificação é codificada como um grafo direcionado (dígrafo) especial, que é então disfarçado em meio ao grafo de fluxo de controle (*control flow graph*) do software em questão. Outros esquemas para *fingerprinting* de software baseados em grafos incluem [3,4,5,6].

Fingerprints baseados em grafos são em geral de difícil análise devido aos efeitos de *aliasing* (maiores detalhes podem ser obtidos em [7,8]). Além disso, *fingerprints* de software estão sujeitos a certos tipos de ataque:

- **adição**, em que o atacante (em nosso cenário, o avaliador malicioso) inclui no código um outro *fingerprint*, na tentativa de ludibriar o algoritmo reconhecedor, que obteria dois *fingerprints* em vez de um;

- **subtração**, em que o atacante descobre a localização do *fingerprint* e simplesmente o remove, evitando assim sua responsabilização por quaisquer vazamentos de código;

- **distorção**, em que o atacante altera sintaticamente o programa por meio de transformações que preservam a semântica do programa original, mas que podem incapacitar a recuperação do *fingerprint* por parte do algoritmo reconhecedor, uma vez que algoritmos reconhecedores baseiam-se sobretudo na topologia do grafo de fluxo de controle do programa, sensível à sintaxe.

- **conluio**, em que dois ou mais avaliadores maliciosos, de posse de programas contendo *fingerprints*, cooperam para descobrir a real localização dos *fingerprints*, possibilitando consequentes subtrações ou distorções.

Vejamos como o protocolo proposto se comporta diante dos tipos de ataque descritos acima. Para o ataque de adição, a necessidade de assinatura pela Autoridade Metrológica impede que o avaliador malicioso insira no código do programa um outro *fingerprint*, pois o mesmo não possui a chave privada da Autoridade Metrológica. Para o ataque de subtração, a codificação do *fingerprint* juntamente com o código da aplicação gera grafos de fluxo de controle em que o grafo do *fingerprint* é indistinguível do grafo do fluxo de controle original do programa. Uma vez que se dificulta assim sua localização, dificulta-se também sua remoção ou

alteração, diferentemente do que se daria caso o dígrafo referente ao *fingerprint* fosse disjunto do grafo de fluxo de controle original, o que poderia despertar a atenção do avaliador malicioso. Para o ataque de distorção, nosso algoritmo de *fingerprint* possui redundância suficiente para se recuperar de ataques que realizem até cinco operações de inserção e/ou remoção de arestas do dígrafo que codifica o *fingerprint*, como extensivamente analisado em [9]. Finalmente, quanto ao ataque de conluio, nosso protocolo por si só não o impossibilita. Contudo, pode-se dificultar a localização do dígrafo que codifica o *fingerprint* por meio do conceito de *diversidade*, pelo qual se aplicam técnicas de transformação de software com preservação semântica para obter diferentes versões de código a serem avaliadas. Dessa forma, dificulta-se enormemente que dois ou mais avaliadores maliciosos possam determinar a localização de seus *fingerprints* por meio de técnicas de *diffing* (comparação de código), pois a diferença entre dois programas não estaria restrita às partes que codificam os *fingerprints*, mas também a trechos de código do próprio instrumento sob avaliação.

A propriedade de não prover fácil localização dentro de um código é comumente chamada de *furtividade* da informação ali embarcada. A codificação do *fingerprint* é realizada de tal forma que os vértices do grafo – ou seja, os blocos do fluxo de controle daquele programa – correspondam a código que será de fato executado. Isso impossibilita que o avaliador malicioso localize o *fingerprint* por meio de ferramentas de *profiling*, as quais coletam dados durante a execução do programa e registram a execução de trechos específicos do código, como instruções e funções, bem como seus tempo de execução.

A geração de um programa acrescido de *fingerprint* possui as seguintes etapas:

(1) Definição, pela Autoridade Metrológica ou TTP, da informação w a ser embarcada no programa original p.
(2) Codificação da informação w na forma de um grafo G.
(3) Geração de trecho de código c cujo grafo de fluxo seja exatamente o grafo G.
(4) Inserção do código c no programa original p, de forma a gerar um programa q, possuidor do *fingerprint.*

A Figura 1 ilustra um exemplo das etapas descritas acima. Para a etapa (2), baseamo-nos no esquema de marcas d'água (*watermarks*) descrito por Chroni e Nikolopoulos [10], que permite codificar uma informação arbitrária (representada por um número binário) por meio de um grafo de fluxo redutível. As arestas de tal grafo (na realidade, um dígrafo) podem ser particionadas em arestas *diretas*, constituindo um caminho hamiltoniano que é único naquele dígrafo, e arestas *de retorno*, que estão representadas por linhas curvas na Figura 1.

Após a definição do dígrafo que codifica o *fingerprint*, é necessário gerar código cujo fluxo de controle corresponda exatamente àquele dígrafo, embarcando-o a seguir no programa original (passos 3 e 4). Para isto, utiliza-se um *embedder* que cria, a partir do dígrafo fornecido, o código que será inserido no programa do instrumento de medição a ser avaliado. O código obtido é então imiscuído no software *ao longo* do código original do instrumento de medição, dado que a inserção do código em uma região disjunta do código original da aplicação seria, como já mencionado, pouco furtiva.

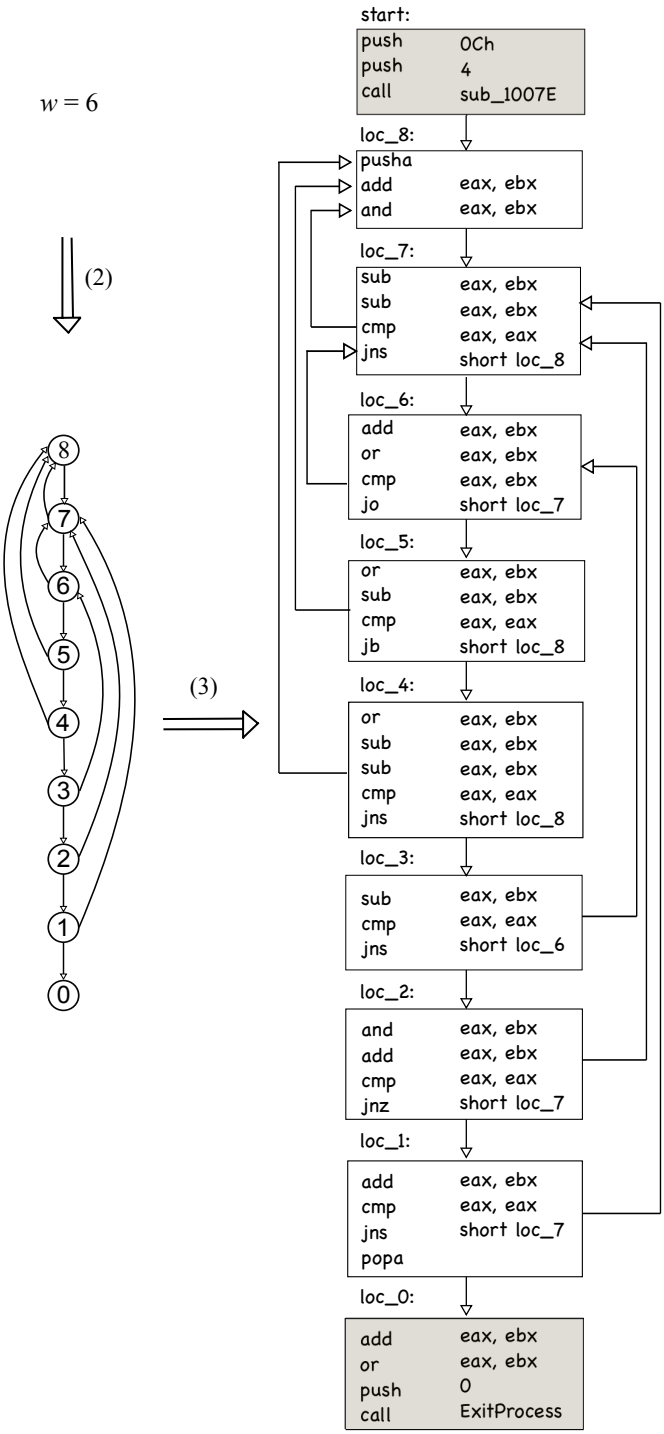

Figura 1. Processo de inserção de *fingerprint*.

A construção do código pelo algoritmo de inserção, que é feita em linguagem *assembly*, baseia-se na inserção de instruções irrelevantes – ou seja, instruções que, embora executadas, não impactam na semântica da aplicação – para a composição de novos blocos de fluxo de controle. Cada um desses blocos, criado com número variável e aleatório de instruções para evitar ataques baseados em reconhecimento de padrões (*pattern-matching*), corresponde a um vértice do dígrafo que codifica o *fingerprint* sendo embarcado.

Instruções condicionais, em particular, são necessárias para representar as arestas de retorno de tal dígrafo. Cada uma dessas instruções condicionais, no entanto, será forçosamente avaliada de forma a direcionar o fluxo da execução para o bloco (vértice) seguinte no caminho hamiltoniano, isto é, a aresta indicando o caminho a ser seguido será sempre a aresta direta com origem no bloco que está sendo executado, e nunca a aresta de retorno com origem naquele bloco[1]. Evita-se, assim, a criação de laços infinitos.

Para possibilitar um conjunto maior de instruções no programa, bem como identificar o início e o fim do *fingerprint,* são inseridas no primeiro e último vértice as instruções PUSH_A, que empilha todos os registradores, e POP_A, que os desempilha. A inserção dessas instruções não afeta a semântica do restante do programa.

O reconhecimento do *fingerprint* segue o caminho inverso da codificação. A partir do programa modificado *q,* e sabendo-se a posição onde foi inserido o *fingerprint*, retira-se o trecho de código *c* cujo grafo de fluxo codifica a informação *w* gerada pela Autoridade Metrológica ou TTP. A recuperação do grafo de fluxo associado a *c* segue algoritmos clássicos (ver [11]). Para obter a informação *w* a partir do grafo de fluxo, utilizamos o algoritmo de decodificação descrito detalhadamente em [9]. Para obter o programa original *p* a partir do programa alterado *q*, é suficiente remover as instruções referentes à marca d'água. Finalmente, a partir do programa *p* e das informações contidas em *w*, é possível comprovar, via chave pública do TTP, quem foi o avaliador responsável pela avaliação daquele código.

III. Considerações Finais

O presente trabalho apresenta um protocolo que permite identificar unicamente o código de programas entregues a terceiros para avaliação, possibilitando a responsabilização de avaliadores, no caso de um eventual vazamento de código. A identificação é feita através do uso de uma técnica de *fingerprinting* baseada em grafos, e o protocolo proposto apresenta resiliência a cenários de ataque por adição, remoção e modificação, podendo, ainda, ser adaptada de modo a apresentar resistência a ataques de conluio.

[1] Para este fim, antes de cada uma dessas instruções condicionais, é inserido um *predicado opaco,* que consiste de instruções cuja execução produz resultados conhecidos de antemão pelo programador, porém de difícil análise por um especialista ou analisador automático.

Referências

[1] R. L. Davidson and N. Myhrvold (1996), "Method and system for generating and auditing a signature for a computer program", US Patent 5.559.884, Microsoft Corporation.

[2] R. Venkatesan, V. Vazirani, S. Sinha (2001), "A graph theoretic approach to software watermarking", 4th International Information Hiding Workshop, pp. 157–168.

[3] C. Collberg, A. Huntwork, E. Carter, G. Townsend and M. Stepp (2009), "More on graph theoretic software watermarks: implementation, analysis and attacks", vol. 51, no. 1, pp. 56–67.

[4] C. Collberg and C. Thomborson (1999), "Software watermarking models and dynamic embeddings", Proc. 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages, POPL'99, pp. 311–324.

[5] C. Collberg, C. Thomborson and G. M. Townsend (2007), "Dynamic graph-based software fingerprinting", *ACM Transactions on Programming Languages and Systems*, vol. 29, no. 6, pp. 1–67.

[6] J. Zhu, Y. Liu and K. Yin (2009), "A novel dynamic graph software watermark scheme", 1st Int'l Workshop on Education Technology and Computer Science 3, pp. 775–780.

[7] R. Ghiya and L. J. Hendren (1996), "Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C". Proc. of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 1–15.

[8] G. Ramalingam (1994). "The undecidability of aliasing". *ACM* Trans. Program. Lang. Syst., vol. 16, no. 5, pp. 1467–1471.

[9] L. M. S. Bento, D. Boccardo, R. C. S. Machado, V. G. Pereira de Sá, J. L. Szwarcfiter (2013), "Towards a provably robust graph-based watermarking scheme", http://arxiv.org/abs/1302.7262

[10] M. Chroni and S.D. Nikolopoulos (2011), "Efficient encoding of watermark numbers as reducible permutation graphs", http://arxiv.org/abs/1110.1194

[11] F. Nielson, H. R. Nielson, C. Hankin (2004), "Principles of Program Analysis". Springer Verlag.