SAND2011-2348C

On Source Code Transformations for Steganographic Applications

Geoffrey Hulette
Computer and Information Science Department
University of Oregon
Eugene, OR, USA
ghulette@cs.uoregon.edu

John Solis
Scalable Computing R&D Department
Sandia National Labs
Livermore, CA, USA
jhsolis@sandia.gov

***Abstract*—The amount of source code publicly available on the Internet makes it attractive as a potential message carrier for steganographic applications. Unfortunately, it is often overlooked since preserving semantics while embedding information in an undetectable way is challenging. To encourage discussion of new and useful techniques, we investigate term rewriting transformations as a method for embedding messages into program source code.**

We elaborate on several possible transformation strategies and discuss how they can be applied in a general steganographic setting. We continue with a discussion on (a) the implications and trade-offs of preserving original semantic properties, (b) the relationship between messages and transformations, and (c) how to incorporate existing natural language processing techniques. The goal of this work is to elicit constructive feedback and present ideas that stimulate future work.

***Keywords*-steganography; source code transformations; term rewriting;**

I. Introduction/Motivation

The field of steganography studies techniques for embedding information into inconspicuous carriers such that outside observers cannot easily detect the presence of this information. One typical application scenario is the discreet transfer of secret messages between two parties. For example, in a country forbidding anti-government rhetoric, its citizens could communicate by transmitting sensitive messages embedded in image files. A second scenario, document watermarking, is a digital rights management method for determining if copies of media are unauthorized.

In both scenarios, the same steganographic techniques can, in principle, be applied to embed watermarks or secret messages, e.g., information is embedded into bitmapped images by manipulating the low-order bits of each pixel. Beyond the image example, techniques exist for a wide range of carriers including: video [1], audio [2], [3], natural language text [4], and binary executables [5].

One document type often overlooked as a potential steganographic carrier is *source code*. This oversight is partly due to the challenge of preserving semantic correctness while simultaneously making the embedded information difficult to detect. For instance, adding dummy methods never referenced by the original source preserves program semantics, but will likely be identified as suspicious by an analyst. Why, therefore, is source code an attractive carrier?

The number of publicly available source code repositories has exploded with the advent of the Internet and open source software. The total amount of available source code is staggering once we consider everything from large repository websites, e.g., SourceForge [6], to individually maintained repositories and instructional websites including sample source code. PlanetSourceCode.com alone claims a database containing over 29 million lines of code [7]. This makes source code an attractive information carrier because it is likely to be overlooked or dismissed by analysts – especially if the communicating users have a computer science or programming background.

In this paper, we investigate some novel techniques for embedding messages into program source code using term rewriting transformations. We look into three general approaches to encoding the transformation and show how each could potentially be applied in a steganographic setting. We continue with a discussion on preserving semantic properties, the relationship between messages and transformations, and how these techniques can be used in conjunction with existing natural language processing techniques. We conclude with some ideas for potential future work.

II. Term Rewriting

Our method encodes a secret message in a log of transformations applied to a given source code text. Crucially, the transformation must be deterministic, so that the message can be reliably recovered. At the same time, the transformation must be flexible enough to admit encodings of a wide variety of messages. *Term rewriting* [8] gives a convenient framework and theory for specifying program transformations. Term rewriting systems are flexible enough for our purposes, and can be restricted to deterministic transformations.

A *term* is an n-ary tree structure representing, in this case, a program's source code. Each node is labeled with a *constructor*, and may have zero, one, or more children. Terms are versatile data structures, and are particularly good for encoding abstract syntax trees. For a programming language like C or Java, valid terms should be well-defined

with respect to the language syntax. In general there may be many different ways to construct a term encoding for a given language – deciding on a consistent and comprehensive term representation is typically the first step in any term rewriting exercise.

Term *rewriting* is a procedure for transforming one term into another. Primitive rewriting transformations are called "rules." A rule contains two components – the first is a pattern, which is matched against the top-level (outermost) term, and the second is another term which is used to replace the input term. Rule application may fail if the pattern does not correspond to the input. If the application matches, the rule application succeeds, and the second component of the rule is the result of the transformation. Primitive rules will typically allow variables in the pattern, which are bound as a side-effect of a successful match and can be used in the rule's result term.

Given a set of primitive rules, we must then provide a *strategy* to apply them. In particular, since primitive rules apply only at the outermost term, we need a way to apply them to sub-terms. We also may want to control how many times to apply a rule in a given term, or other aspects of the transformation. The most general strategy is to apply rules non-deterministically, as many times as possible, and at every sub-term where they may apply; this strategy is called *reduction*. Reduction is problematic for stenography because it may be non-terminating and non-deterministic in general. We suggest the use of explicit, deterministic strategies created using rule combinators. An example of one such set of combinators follows.

A. System S

System S [9] is a core language for term rewriting strategies. It provides a number of simple combinators that allow primitive rules to be composed into complex program transformation functions. For example, the *sequence* combinator allows us to apply two rules in sequence, with the second rule acting on the output of the first. Similarly, the *left choice* combinator will attempt to apply the first rule, and only try the second rule if the first rule fails. Crucially, System S permits us to preclude non-deterministic strategies by eschewing the pure *choice* combinator, as described in [9].

To support steganographic applications, we extend System S with tracing semantics to record rule applications. First, we assert that primitive rules must be labeled. Second, successful application of a primitive rule is logged (i.e. its label recorded) sequentially, in the order of application. The labels of the rules thus form the alphabet for the obfuscated message, and the trace itself forms the message string. We omit the formal semantics for lack of space.

System S treats failure as a special case. It is possible for a given program in System S to fail, indicating that no rules were successfully applied and the program was not transformed. All failures in System S are represented by a single distinguished token in lieu of a transformed program, and there is no trace output. For the purposes of steganography failure could be considered a valid message, albeit with exactly one form.

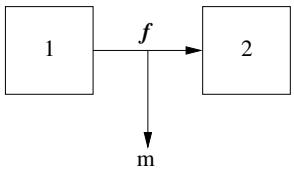


Figure 1. One-way transform

III. Methods

Now we will examine some ways in which the above machinery might be put to steganographic use. We consider three separate scenarios. In each of following scenarios, let F, G, etc. represent term rewriting programs in System S as described above, let x, y, etc. range over term encodings of some program source code, and let m, n etc. range over trace outputs. We write $x \xrightarrow{F} (x', m)$ to say that the application of rewriting program F to the term representation of source code x is successfully rewritten to another term x' with trace m. We write $x \xrightarrow{F} \downarrow$ to indicate that the transformation failed.

A. One-way transform

In the first scenario, we construct a transformation F and a source code term x such that $x \xrightarrow{F} (x', m)$ where x' is a valid transformed program and m is the desired obfuscated message. We expect that F would be communicated between sender and recipient through some back channel, and that the source code x would be available publicly or transmitted through some low security channel. To recover the message the recipient simply evaluates the transformation. See Figure 1 for a graphical illustration.

This approach is conceptually simple, but has the drawback that F must be communicated separately.

B. Two-way transform

In the second scenario, we construct a transformation F and term x as before, such that $x \xrightarrow{F} (x', m)$ where x' is a valid transformed program and m is the desired message. We add the constraint that there must exist a transformation F^{-1} such that $x' \xrightarrow{F^{-1}} (x, m^{-1})$, where m^{-1} is the mirror (reversed) string m. Conceptually, F^{-1} is the inverse transformation of F. Under certain conditions and with some restrictions (beyond the scope of this paper, and a topic of ongoing research), given F we can produce F^{-1} or vice versa.

The advantage of this approach over the first is that the original source code need not be distributed. Instead, just the transformed code x' can be publicly distributed, and the

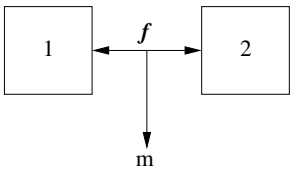
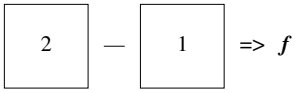


Figure 2. Two-way transform



$f(1) = \mathrm{m}$

$f(2) = \mathrm{m}$

Figure 3. Recover transform function

message recovered by transforming it with F^{-1} and then simply reversing the resulting trace to recover the message (see Figure 2). This may be desirable in cases where, for whatever reason, the original source code represents sensitive information. In this case, our method may be used to obfuscate the original source code in addition to the message itself.

C. Recover F from differences

In our third and final scenario, the transformation F is not communicated, but instead is recovered by examining the differences between two terms x and x' constructed so that $x \xrightarrow{F} (x', m)$ (see Figure 3). Notice that in this case, F must be unique – that is, if $x \xrightarrow{F} (x', m)$ and $x \xrightarrow{G} (x', n)$ then we require that $F = G$ (and, consequently and crucially, that $m = n$).

The advantage of this approach is that there is no need for a second channel to communicate F, since it can be recovered from source code alone and then used to transform x and recover m.

The difficulty of recovering F from x and x', as well as the restrictions involved, is a topic of ongoing research. This scenario is therefore presented mainly for theoretical interest.

IV. Discussion

A. Semantic Properties

In some cases it may be useful to preserve the semantics of programs under transformation. This would be desirable, for example, if we foresaw the source code being inspected for legitimacy – in this case, preserving or mostly preserving the semantics of the code could make it appear that the transformation was applied in the service of software development rather than steganography. Conversely, a program that has been transformed in such a way as to render it broken, inefficient, or nonsensical may invite unwanted scrutiny.

Ensuring full preservation of program semantics under transformation is quite difficult, even without ulterior steganographic motives. Adding the constraint that the transformation must also induce a particular message makes the problem even more so. We expect that this will be an interesting area for future work.

B. Constructing Messages

We have not fully addressed the question of how to construct a transformation F and source code x so as to induce a particular, desired message m. This might be especially difficult if we were given a fixed value for either x or F in addition to m and asked to provide the single missing element. We expect this case could be a common one – for example if we wanted to embed a message in an existing, publicly available code. Even if we are allowed to choose both F and x, we currently do not have a procedure to construct them given m. This is an area of ongoing work.

C. Combining with Natural Language Processing

The methods discussed above naturally lend themselves to being combined with natural language processing (NLP) techniques. The Semantilog Project [10] is a comprehensive bibliography of linguistic steganographic techniques, both theoretical and applied, for a number languages, including English, Japanese, Chinese, Persian, and Arabic. In our context, we treat NLP steganography as a black box capable of embedding information into the comments and other documentation found in source code files. The specific instantiation, which may be based on a particular programming language or documentation style, is irrelevant.

The basic idea is to use existing NLP steganography tools to create a second information channel in the source text. Combined with the term rewriting information channel, we can now use the two separate channels to (a) duplicate the embedded message or (b) apply cryptographic secret sharing techniques to split the message.

The first approach improves the robustness of message recovery. Ideally, the techniques applied to each individual channel will already be fairly robust, i.e., small changes do not affect the ability to recover messages. However, having a second information channel allows us to recover the embedded message when the first channel is manipulated or tampered beyond recovery, e.g., by removing all comments or complete restructuring of source code.

In the second situation, the goal of cryptographic secret sharing is to make the source code robust against statistical analysis. A statistical analysis, looking for variations in entropy across the source program, will not recover any additional information since the secret share is itself indistinguishable from random. This will be true for both of the information channels present in the source code. An alternate strategy is to make the source code documentation completely separate from the source code itself. In this

situation, an analyst must correctly identify and associate the documentation file with its corresponding source code. Note that this mapping can (and should be) independent and random, e.g., the documentation for source file X is mapped to source code Y. This mapping can be identified using a pseudo-random permutation keyed by a shared secret key, established a-priori, between the communicating parties.

V. Related Work

The field of steganography has been well studied over the course of its existence. Early results in this field [11], [12] focused on identifying models and terminology, along with theoretical limits of information embeddable within a given carrier. Later results focused on techniques for embedding information into a specific carrier, including video [1], audio [3], natural language text [4], and binary executables [5].

The most closely related area is the sub-field of *software watermarking*. These techniques discourage illegal duplication by allowing authorities to prove ownership. This can be accomplished through register allocation patterns [13], dynamic path execution [14], graph-based approaches [15], and spread-spectrum techniques for robust watermarks [16]. These approaches, applicable to executable code, are not concerned with the originating source code as a carrier.

Our work differs from previous approaches in that we use source code as the carrier. In contrast to watermarking, we are not interested in detecting illegal duplication. Instead, we would like to investigate how source code, and more specifically source code transformations, can be used to embed secret messages or information.

VI. Future Work and Conclusion

Clearly, there is considerable work to be done. The immediate next step is to implement a system that incorporates the basic capability described in Section III-A. Such an implementation would allow us to answer open questions, such as, what is the total amount of information each transformation is capable of embedding? A concrete implementation would also help determine the feasibility of incorporating existing NLP steganographic approaches as a second information channel. This may turn out to be impossible if the information capacity of one channel greatly exceeds that of the other.

The most challenging step will be to develop the theory to support invertible term rewriting transformations and to recover unique functions from differences between source codes. This theory will be a prerequisite for the application scenarios described in Sections III-B and III-C. It may be difficult or impossible to guarantee that a transformation is invertible, or that the inversion is unique. However, the three approaches we have identified merit further investigation. We hope that this paper elicits constructive feedback and stimulates future work in this area.

References

[1] H. Noda, T. Furuta, M. Niimi, and E. Kawaguchi, "Application of BPCS steganography to wavelet compressed video," in *ICIP '04*, vol. 4, Oct 2004, pp. 2147–2150.

[2] K. Gopalan, "Audio steganography using bit modification," in *ICME '03*, vol. 1, July 2003, pp. 629–632.

[3] N. Cvejic and T. Seppanen, "Increasing robustness of LSB audio steganography using a novel embedding method," in *ITCC '04*, vol. 2, April 2004, pp. 533–537.

[4] M. J. Atallah, V. Raskin, M. Crogan, C. Hempelmann, F. Kerschbaum, D. Mohamed, and S. Naik, "Natural language watermarking: Design, analysis, and a proof-of-concept implementation," in *IHW '01*. Springer-Verlag, 2001, pp. 185–199.

[5] R. El-Khalil and A. D. Keromytis, "Hydan: Hiding information in program binaries," in *Information and Communications Security*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, vol. 3269, pp. 287–291.

[6] "SourceForge.net: Find, create, and publish Open Source software for free," http://www.sourceforge.net/.

[7] "PlanetSourceCode.com: The largest public source code database on the internet," http://www.planet-source-code.com/.

[8] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge University Press, 1998.

[9] E. Visser and Z. el Abidine Benaissa, "A core language for rewriting," *Electronic Notes in Theoretical Computer Science*, vol. 15, pp. 422–441, 1998.

[10] R. Bergmair, "A comprehensive bibliography of linguistic steganography," The Semantilog project: http://www.semantilog.org/biblingsteg/.

[11] R. Anderson and F. Petitcolas, "On the limits of steganography," *Selected Areas in Communications, IEEE Journal on*, vol. 16, no. 4, pp. 474–481, 1998.

[12] C. Cachin, "An information-theoretic model for steganography," in *Information Hiding*, ser. Lecture Notes in Computer Science. Springer, 1998, vol. 1525, pp. 306–318.

[13] G. Myles and C. Collberg, "Software watermarking through register allocation: Implementation, analysis, and attacks," in *ICISC 2003*, ser. Lecture Notes in Computer Science. Springer, 2004, vol. 2971, pp. 274–293.

[14] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp, "Dynamic path-based software watermarking," *SIGPLAN Not.*, vol. 39, pp. 107–118, 2004.

[15] R. Venkatesan, V. V. Vazirani, and S. Sinha, "A graph theoretic approach to software watermarking," in *IHW '01*. Springer-Verlag, 2001, pp. 157–168.

[16] J. P. Stern, G. Hachez, F. Koeune, and J.-J. Quisquater, "Robust object watermarking: Application to code," in *IH '99*. Springer-Verlag, 2000, pp. 368–378.