

Loco: An Interactive Code (De)Obfuscation tool

Matias Madou Ludo Van Put Koen De Bosschere
Ghent University, Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium
{mmadou,lvanput,kdb}@elis.ugent.be

Abstract

This paper presents LOCO, a graphical, interactive environment to experiment with code obfuscation and deobfuscation transformations, which can be applied automatically, semi-automatically and by hand. LOCO is an extension of the multi-platform visualization tool LANCET, combined with an obfuscation infrastructure in the underlying link-time program rewriter DIABLO. By use of LOCO, a developer can easily navigate through the control flow graph of a program and do fine-grained obfuscation, test new obfuscation transformations, test the robustness of existing transformations or improve existing transformations.

Keywords code obfuscation, security, binary rewriting

1. Introduction

In 2004, the world spent more than \$59 billion on commercial packaged PC software, while software worth over \$90 billion was actually installed¹. This means that the software industry loses \$31 billion of revenue due to piracy despite several protection mechanisms in commercial software packages. Crackers use several techniques to eliminate the protection mechanisms built into the software, for example they bypass the protection mechanism and redistribute the modified program.

A commonly used software protection mechanism is the use of a unique software license key that will be validated by a license key validation algorithm. This algorithm checks a property that a valid license key should obey. Cracking this mechanism can be done in three ways. The first possibility is by using a widespread, stolen license key. The second possibility is to bypass the validation algorithm by modifying the software. Lastly, the validation mechanism can be reverse engineered to build a license key generator. While the first two cracking methods are the simplest, they might impose some limitations on the use of the cracked software. In the case the software exchanges its license key with external programs, no valid license key can be provided since either there is no license key or the widespread license key is rejected because it has been black-listed. An example of this is a failure to automatically update the cracked software from a server. The last cracking mechanism has no such inconveniences and poses a more severe threat to software

¹Second annual BSA and IDC global software Piracy Study. www.bsa.org/globalstudy

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PEPM '06 January 9–10, Charleston, South Carolina, USA.
Copyright © 2006 ACM 1-59593-196-1/06/0001... \$5.00.

protection. This is illustrated in the case of Microsoft XP, where service pack 2 can't be installed with a widespread license key, but programs can be found on the Internet that produce valid license keys.

An effective solution to protect a validation algorithm is obfuscating the program. The goal of code obfuscation is making it harder for a cracker to understand and reverse engineer the program. This is done by transforming the program into a new one, while maintaining its functionality. As it is difficult to measure the programmers effort to undo an obfuscation transformation, or to understand the obfuscated algorithm, there is a need for an experimental environment where obfuscation and deobfuscation transformations can be tested interactively.

We present LOCO, an experimental environment where code obfuscation and code deobfuscation transformations can be applied automatically, semi-automatically and by hand. LOCO is an extension of the graphical user interface LANCET[8], combined with an obfuscation infrastructure in the underlying link-time program rewriter DIABLO[5]², which allows us to do fine-grained code obfuscation. LOCO will be freely available from the DIABLO website.

The remainder of this paper is organized as follows. Section 2 presents DIABLO and LANCET, the underlying library and the graphical user interface. The obfuscation transformations implemented in DIABLO are introduced in Section 3. The interactive code obfuscation and deobfuscation are discussed in Section 4 and 5. Finally, conclusions are drawn in Section 6.

2. Lancet and Diablo

LANCET can visualize the call graph of a program and the control flow graphs (CFGs) of the procedures. Zooming and panning these graphs is possible. Lists of basic blocks and procedures can be searched by sorting them on different properties. Besides navigating the graphs of a program, LANCET also provides means to edit the graphs.

The underlying DIABLO library is a multi-platform link-time binary rewriting framework which we will use with the x86 back-end and ELF object file format. The DIABLO framework performs the task of a traditional linker but currently only handles statically linked binaries. DIABLO needs the same input files as the native linker.

3. Obfuscation transformations

In this section, we describe existing obfuscation transformations that are implemented into the underlying binary rewriter DIABLO.

The most popular control flow obfuscation technique is described by Wang[9] and is called control flow flattening. The idea is that all basic blocks of a function appear to have the same set of predecessors and successors. This obfuscation transformation takes a central part in an industrial obfuscation tool by Cloakware Inc.[2].

²<http://www.elis.ugent.be/diablo>

It is a technique which is not only used for obfuscation, but also for watermarking [2, 10]. This technique was developed to be applied on a function level, but the granularity can be changed. In LOCO this transformation can also be applied on a selected group of basic blocks, that can be in different functions.

Linn and Debray [6] introduce branch functions and apply other control flow transformations with the aim of thwarting the static disassembly of executable code. Jumps are substituted by calls to a branch function which redirect to the original target. This transformation is normally applied to all jumps in the program, but can be done for a selected group of basic blocks in LOCO.

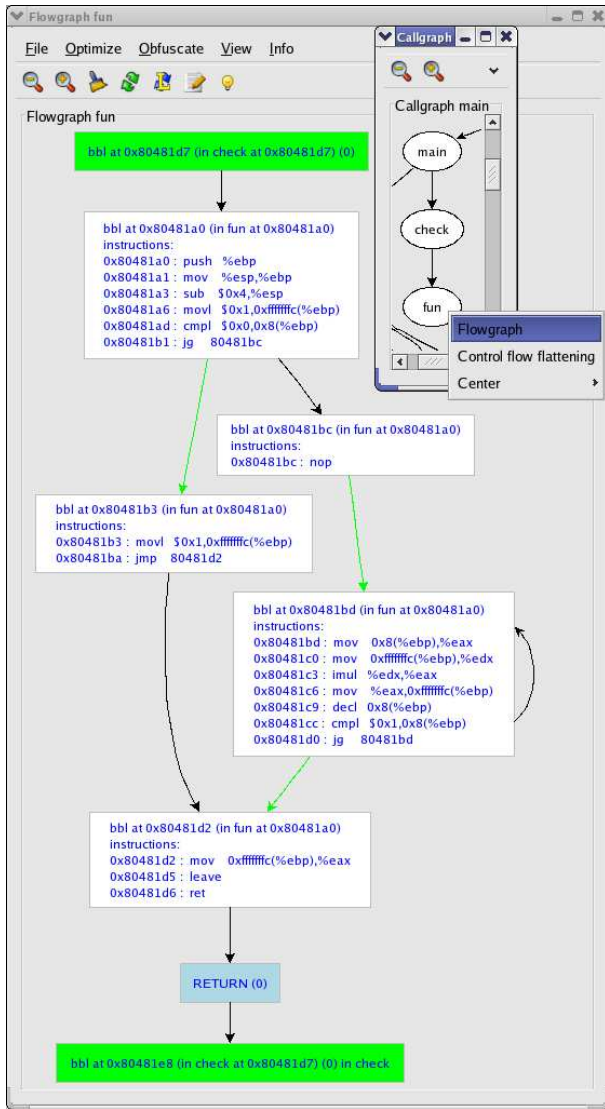


Figure 1. Original CFG of the factorial function *fun*

Another well-known technique for obfuscation [3] (and also for watermarking [1]) is the use of opaque predicates in programs. An opaque predicate exploits a property that is known at obfuscation time, but that is hard to derive afterwards. This property can be used to guide the execution of a program. The fake target of the conditional jump following an opaque predicate is in most cases determined at random. In LOCO we can choose our fake target.

Collberg et al.[4] describes several obfuscation transformations that could intentionally be applied to Java programs. Some of the

transformations, like inserting dead or irrelevant code, extending loop conditions, adding redundant operands, loop transformations and changing encoding, can also be applied during binary transformations and can easily be integrated in LOCO.

4. Interactive code obfuscation

In the following section, we describe some of the advantages of an interactive obfuscation tool.

4.1 Fine-grained code obfuscation

Regular obfuscation tools such as the obfuscation transformations implemented in DIABLO and PLTO[7] apply their transformations on the entire code. Obfuscating uninteresting code only slows down the program execution but it does not increase the level of security. We would like to avoid the obfuscation of uninteresting code and have more control over the selection of program parts where obfuscation transformations should be applied. With fine-grained code obfuscation as provided in a graphical tool, critical program points can be selected and code obfuscation transformations can be applied on functions, a group of basic blocks or selected instructions.

In LOCO, an obfuscation transformation can be chosen from a collection of commonly used obfuscation transformations. The difference with a regular code obfuscator consists of the possibility to obfuscate just the parts we would like to see obfuscated. A user can explore the program and select points that he would like to obfuscate by just a click.

Using a simple example, we will show how LOCO works. In this example, the license key consists of two values, where the factorial of the first value has to be equal to the second value to be a regular license key. The following *check* function will be used to validate the license key:

```

bool check(int key_part1, int key_part2)
{
    if(fun(key_part1)==key_part2)
        return true;
    return false;
}

```

```

int fun(int key)
{
    int a=1;
    if (key<1)
        a=1;
    else
        do{
            a *= key--;
        }while (key>1);
    return a;
}

```

In Figure 1, the CFG of the original factorial function *fun* can be seen. It's clear that following the right path, a loop computing the factorial is executed, while following the left side, the return value is set to 1 and control flows directly to the return.

Figure 2 presents the obfuscated CFG of the factorial function *fun*. Two obfuscation transformations have been applied. First, standard control flow flattening was performed on the entire factorial function. Next, in the basic block that originally contained a loop a true opaque predicate was added. As the target for the (unreachable) false path, the entry block of the function was chosen.

In our example, the entire function *fun* is first obfuscated by use of control flow flattening, but it is also possible to select some basic blocks of this function, combined with basic blocks from other functions and apply the control flow flattening obfuscation

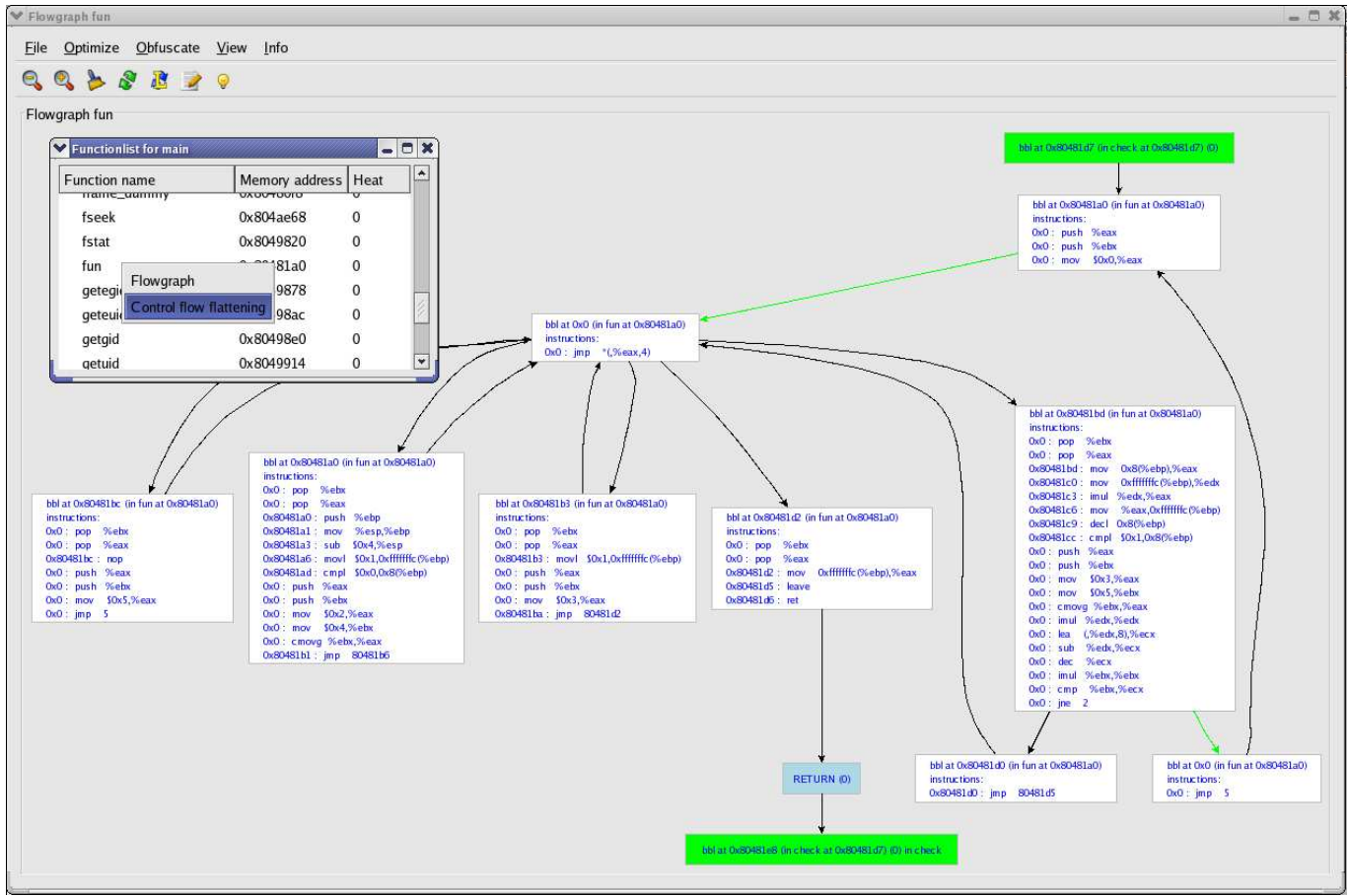


Figure 2. Obfuscated CFG of the factorial function, after two obfuscation transformations have been applied.

on them. It will be harder for a cracker to delimit functions in the program, because basic blocks will be shared by several functions.

In the absence of general obfuscation metric, a software distributor needs to decide himself which obfuscation techniques to use. Using the presented tool, the distributor can try different obfuscation transformations, evaluate the techniques by visual inspection and select the most appropriate transformations.

4.2 Testing new obfuscation transformations

When developing a new obfuscation transformation, it can be hard to predict its effectiveness without first applying and testing it. A developer might want to try out a transformation by hand before starting the implementation of an automatic obfuscation transformation. In that case, the developer saves himself the implementation effort if the result is disappointing. Manual application of a transformation is also useful when an algorithmic transformation turns out to be too complex to implement. In this case, an interface that lets you record and reuse code transformations could be very useful. This functionality is however not provided in LOCO yet.

The graphical program editor LANCET gives the user feedback while he is editing the code. When applying transformations by hand, the program semantics can be broken in which case the developer has to take countermeasures or undo the transformation. LANCET gives for example a warning when a live value is overwritten.

Manual edits can not only be used for testing new transformations, also existing transformations can be hand-optimized or

extended. An automatic obfuscation transformation can be customized, to be more robust against inspection of an attacker. Whereas an experienced attacker might recognize a standard obfuscation transformation, he could be misled when this obfuscation transformation is tuned a little bit.

5. Code Deobfuscation

Evaluating the robustness of an obfuscation transformation is not an easy task. There is no unified metric that gives an answer to the question: How difficult is it to break a given obfuscation transformation? Crackers have to navigate through the code and try to figure out what is happening. It is however more difficult to build a CFG from a statically linked program since most of the information a linker uses is lost in the final binary.

In an ideal case, a cracker can identify the relevant parts of the program and build a CFG of the code. In that case, a tool as the one presented here can help the cracker to discover the functionality of the code. We have added some features to LOCO that help in deobfuscating a program. This can also help the software distributor in the evaluation of the obfuscation techniques that have been applied, by having a fictive, in-house cracker trying to break the obfuscation.

Using LOCO, a cracker/developer can navigate through the code and change edges, basic blocks, insert instructions, ... in order to deobfuscate the code. In addition, the underlying Diablo framework

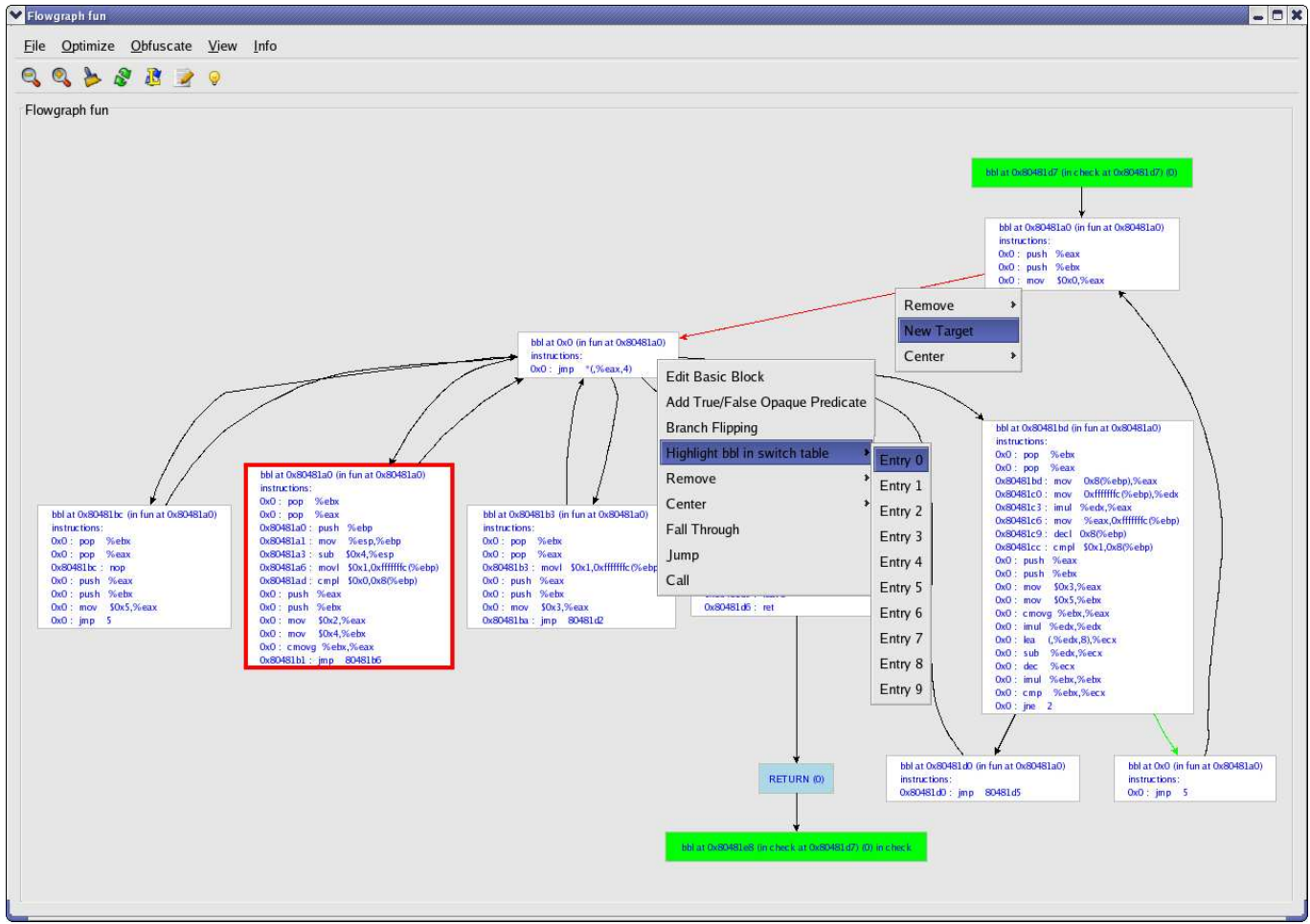


Figure 3. Deobfuscation of the obfuscated factorial function.

provides a set of analysis to extract information from the control flow, like constant propagation and liveness analysis.

A tool like LOCO can also be used for fine-tuning automatic deobfuscation transformations. For example, an automatic deobfuscation transformation for the standard control flow flattening[7] removes 99% of the added edges. With LOCO, a cracker can track edges that couldn't be eliminated. If he's able to identify these edges, this will provide information on the shortcomings in the deobfuscation transformation and he might improve his transformation to eliminate the remaining edges.

Using the previously obfuscated function (Figure 2), we will show the deobfuscation process. Manual inspection of the obfuscated function reveals that control flow falls through from the entry basic block to the switch-block. The entry basic block inserts zero into register `%eax`. Register `%eax` is used as the offset in the switch table. With LOCO, we can highlight the target basic block of the entry on the switch table, which is needed since in a CFG address computations are meaningless. A right click on the switch-block pops up a window where the user can choose an entry in the switch-table to highlight the appropriate basic block. Now we can make changes to the CFG to modify the control flow so that the highlighted basic block becomes a successor of the entry basic block directly.

We have several different possibilities to change control flow. The most efficient one in this case is only changing the target of

the fallthrough edge. One of the choices is choosing a new target out of a popup menu. Other possibilities to change control flow are deleting and adding edges. When deleting an edge, a user can choose to let LOCO handle all side-effects. A side-effect is for example substituting a conditional jump with an unconditional jump when the fallthrough edge following a conditional jump is removed. Adding new edges is done by clicking the head and tail for the new edge.

After a new target is chosen, the deobfuscation process can be continued from the new target of the modified edge. The deobfuscator can figure out that control flow goes two ways, depending on a conditional definition of register `%eax`. As in the previous case, we can modify the CFG such that the switch-construct is bypassed and control flow goes directly to the two possible targets. In order to do so, the last instruction of the basic block has to be modified which can be done using a basic block editing window. As can be seen from Figure 4, instructions in a basic block can be edited and removed. On top of this new instructions can be added and a basic block can be split.

Using the graph and block editing features, the original CFG can be rebuilt. After the control flow has been recovered, the analysis from the underlying DIABLO framework can be used, e.g. liveness analysis to remove useless instructions. Most of these analysis improve the readability of the code, which helps in revealing the functionality.

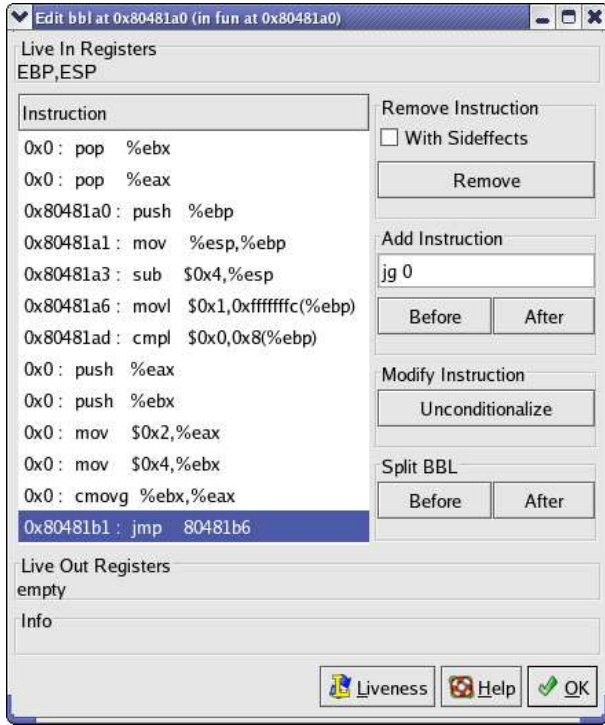


Figure 4. The basic block editing window. When inserting *jmp* instructions, the jump target is determined by the outgoing edges, which explains the meaningless value 0 in the 'Add instruction' field.

After the deobfuscation process, a lot of push and pop instructions will be left in the code. Each basic block that was reachable from the switch instructions starts with two pop-instructions and each basic block that had an outgoing edge to the switch instruction ends with two push instructions. With a simple local optimization these instructions could automatically be removed but to be sure that these instructions were inserted by the obfuscation transformation, liveness analysis and stack analysis should be used. Liveness analysis will also remove the inserted conditional move instructions. After the whole deobfuscation transformation, we end up with exactly the same CFG as we started.

6. Conclusion

We have developed LOCO, a graphical, interactive, easy-to-use experimental environment to test code obfuscation and deobfuscation transformations. We described obfuscation transformations that can be used for different purposes in the LOCO environment, for example to do fine-grained code obfuscation. LOCO can also be used to make existing code obfuscation transformations more robust against attacks, or to find out how easy they can be broken. With LOCO it is possible to interactively test new obfuscation transformations and try to deobfuscate them before implementing them in an automatic transformation. The tool described can be used as an experimental environment for code obfuscation transformations, which is a big advantage in the battle against software piracy.

Acknowledgments

The authors would like to thank the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) and the Fund for Scientific Research Flanders (FWO) for their financial

support. This research is also partially supported by Ghent University and by the HiPEAC network

Information

LOCO is based on DIABLO and is freely available from the DIABLO web site³. The DIABLO manual available on the web site covers installation of the tool and provides information on how to add or modify DIABLO and LOCO functionality.

References

- [1] G. Arboit. A method for watermarking java programs via opaque predicates. In *Proc. of ICECR-5*, October 2002.
- [2] S. Chow, Y. Gu, H. Johnson, and V. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *G. Davida and Y. Frankel, editors, Information Security, ISC 2001*, volume 2200 of *Lectures Notes in Computer Science (LNCS)*:Springer-Verlag, 2001. 68, 2001.
- [3] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, The University of Auckland, New Zealand, 1997.
- [4] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, pages 184–196, 1998.
- [5] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere. Link-time optimization of ARM binaries. In *Proc. of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2004.
- [6] C. Linn and S. K. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. 10th ACM Conference on Computer and Communications Security (CCS 2003)*, Oct 2003.
- [7] S. Udupa, S. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering (WCRE 2005)*. IEEE Computer Society, November 2005.
- [8] L. Van Put, B. De Sutter, M. Madou, B. De Bus, D. Chanet, K. Smits, and K. De Bosschere. Lancelot: A nifty code editing tool. In *Proc. 6th ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM Press, 2005.
- [9] C. Wang. *A Security Architecture for Survivability Mechanisms*. PhD thesis, Department of Computer Science, University of Virginia, October 2000.
- [10] K. S. Wilson and J. D. Sattler. Software control flow watermarking, Aug 2004. Baker and Botts, US2005/0055312 A1.

³ <http://www.elis.ugent.be/diablo/obfuscation>