# Poster: Design and Automatic Evaluation of Control Flow Obfuscators in a Dynamic Attack Context

Geoffroy Gueguen
PhD Student, CIDRE Research Group,
SUPELEC/Inria/IRISA (France)
geoffroy.gueguen@gmail.com

Sébastien Josse
DGA (French Ministry of Defense)
sebastien.josse@polytechnique.edu

Ludovic Mé
Professor, CIDRE Research Group,
SUPELEC/Inria/IRISA (France)
ludovic.me@supelec.fr

Software protection against reverse engineering has become a subject of interest for security researchers. *Obfuscation* transformations are designed to increase the cost of information extraction, through *data flow* and/or *control flow* transformations.

When designing a protection method, you have to pay attention to both its correctness, its impact on the program performances and its resilience against static and dynamic analysis tools[1] commonly used by attackers. You will find in the literature many techniques for hiding both data and control flow, along with evidence of their resilience against static analysis. Experience shows that many of them do not provide acceptable security when assessed by analysts in the real world, using a conjunction of static and dynamic analysis tools.

We address this problem by proposing:

- A candidate algorithm, designed to be resilient against both static and dynamic attacks.
- A realistic evaluation of its resilience, through the use of an automatic deobfuscation tool, using a conjunction of static and dynamic analyses.

## 1. Control flow obfuscation

To be resilient in both static and dynamic attacks contexts, an obfuscation transformation has first to be resilient against static analysis algorithms. We build a new obfuscation transformation upon an existing control flow transformation, proved to be resilient in a *static attack context*.

### 1.1. Control Flow Flattening (CFF)

The goal of the *Control Flow Flattening* obfuscation [CT02] method is to force an adversary to perform global analysis to understand local control flow transfers. Both forward and backward analyses are obstructed. However the CFF protection mechanism alone can be inversed, by applying suitable static optimization passes [UDM05]. To thwart such attack methods, it is required to strengthen the CFF mechanism by embedding a “difficult problem” in the compilation process to thwart static analyses such as constants or ranges propagation, etc.

### 1.2. Strengthened Control Flow Flattening (SCFF)

In [CP10], a protection scheme (figure 1) is proposed to strengthen the CFF obfuscation transformation, by using a cryptographic hash function. This protection scheme is designed to obstruct flow-sensitive static analyses, which rely on accurate control flow information.

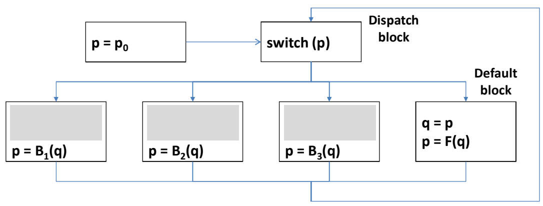


Figure 1: Strengthened CFF

The initial value ($p$=$p_0$) is used by a *dispatcher block* to synchronize the execution of the *basic blocks*. Each basic block ends with a call to the B function. A *default block* is executed per default in the switch-case loop. This default block updates the state $p$ variable with a call to the hash function F.

This protection scheme is proved to be statically secure under the assumption that the *initial value* setting, which is done by *opaque predicates* concatenation, remains secret.

If such an assumption is valid in a static attack context, it does not hold in a dynamic attack context. Indeed, by tracing the execution flow of the program, an attacker is able to get both the truth value of the opaque predicates vector and to obtain the effective ordering of basic blocks. By this way, a dynamic

[1] Among these tools, you will find some static disassembler, debugger, system level diagnostic tools, binary instrumentation tools, but also some more specialized tools using hybrid static / dynamic methods.

abstract interpreter is able to recover easily most of the control flow information.

### 1.3. Parallel Control Flow Flattening (PCFF)

To overcome this limitation, we propose the following key idea: to fork each basic block as independent processes (figure 2). The main process enters a debugging loop after having created its child processes. When it receives a signal from one of the processes, it updates the state of all of them. Each process embeds control instructions and executes a switch loop, which ends with a call to the B function.

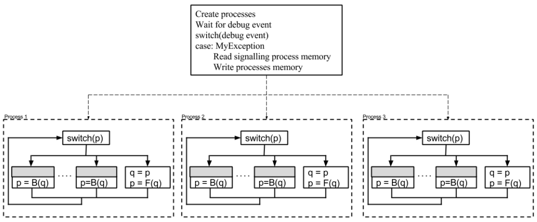


Figure 2: Parallel CFF

A dynamic abstract interpreter cannot guess the order of the basic block execution, because each of them is executing simultaneously / concurrently. Moreover, current dynamic analysis tools are not adapted to trace simultaneously in a coherent way several parallel processes exchanging signals and data.

## 2. Evaluation

A common way to implement obfuscation transformation is to specialize an existing compilation chain, by adding some obfuscation passes in the compilation stages. This is done in the same way as optimization passes are added, by working on one of the intermediate representations of the program being compiled.

We have used the LLVM [LA04] compilation framework to implement the CFF, ECFF and Parallel CFF (PCFF) obfuscation transformations.

### 2.1. Dynamic Abstract Interpreter: towards a more realistic model of the attacker

A current trend in reverse analysis is to try to undo obfuscation transformations, by using binary rewriting tools, which can be seen as specialized compilation chains, using binary front-ends instead of source languages front-ends. Abstract interpreters provide an interesting way to model such an attacker.

Deobfuscation passes must be representative of the many methods used by an attacker, either static (partial evaluation, slicing, symbolic execution) or dynamic (tracing, concolic execution). Observable dynamic semantics are used to specify dynamic abstract domains [Jos09]. Let us call *Dynamic Abstract Interpreter* an abstract interpreter using dynamic analysis.

### 2.2. Preliminary results and future work

We have used the normalization module of VxStripper [Jos14] to implement a dynamic abstract interpreter. This tool is based on the dynamic binary translator engine of QEMU [Bel05] and on the LLVM compilation chain.

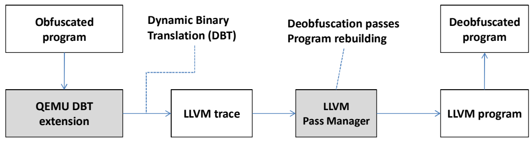


Figure 3: Normalization module

Well-chosen optimizations used in conjunction with the partial evaluation induced by the dynamic translation of target binary code to its LLVM representation are sufficient to recover automatically the control flow when hidden by CFF and ECFF obfuscation passes. In the contrary, as there is no dynamic abstract interpreter able to handle simultaneously several processes contexts to date, PCFF cannot be defeat currently. As a future work, we will investigate this challenge.

## 3. References

[Bel05] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator", in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41-46, 2005.

[CP10] J. Cappaert and B. Preneel, "A general model for hiding control flow", in *Proceedings of the tenth annual ACM workshop on Digital rights management*, 2010, pp. 35-42.

[CT02] C. Collberg, C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection, Software Engineering", in *IEEE Transactions on Software Engineering*, vol. 28, pp. 735-746, 2002.

[Jos09] S. Josse, "Dynamic analysis and detection of viral code in a cryptographic context", *PhD Dissertation, Ecole polytechnique*, 2009.

[Jos14] S. Josse, "Malware Dynamic Recompilation", in *IEEE Proceedings of the 47th HICSS Conference*, 2014.

[LA04] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation", in *International Symposium on Code Generation and Optimization*, pp. 75--86, 2004.

[UDM05] S. K. Udupa, S. K. Debray, M. Madou, "Deobfuscation: Reverse Engineering Obfuscated Code", in *Proceedings of the 12th Working Conference on reverse Engineering*, pp. 45-54, 2005.