Journée du GT Méthodes Formelles et Sécurité

30 janvier 2020
Paris 6, campus de Jussieu

Organisation :
Karine Heydemann, Damien Couroussé

Reponsables du GT :
Sébastien Bardin, Stéphanie Delaune

EXCLUSIF !!

Cristian Cadar	Titre à venir.
Barbara Fila	Attack trees: meaning, analysis, and correctness
Itsaka Rakotonirina	Efficient verification of observational equivalences in finite-process calculi.
Cătălin Hriţcu	When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise.
Thomas Letan	Specifying and Verifying Hardware-based Security Enforcement mechanisms.
Patricia Mouy	Use of formal methods in the Common Criteria Context.
Johan Laurent	A Cross-Layer Security Approach: Combining Accurate Modelling of Hardware Faults with Static Software Analysis.

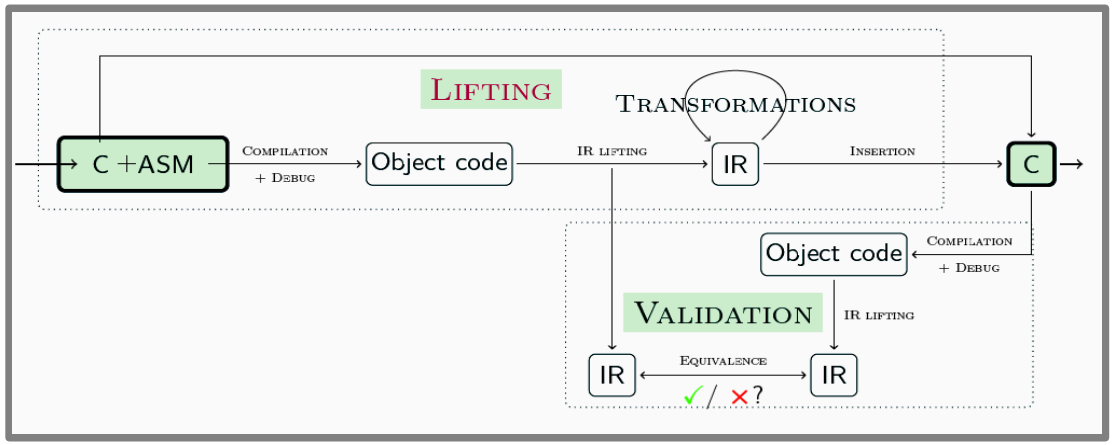
Pensez à vous inscrire !! https://gtmfs2020.github.io/

Breaking news 1
Inline assembly & verification (ASE 2019)

BINSEC

By Frédéric Recoules

- Problem = verifying mix code
- Standard verif tools do not work
- Solution : verification-oriented lifting
- Evaluation :
  . 2000+ asm chunks from Debian
  . Frama-C, Klee



Get rid of inline assembly through verification-oriented lifting

Frédéric Recoules*, Sébastien Bardin*, Richard Bonichon*, Laurent Mounier† and Marie-Laure Potet†
*CEA LIST, Paris-Saclay, France
firstname.lastname@cea.fr
†Univ. Grenoble Alpes. VERIMAG, Grenoble, France
firstname.lastname@univ-grenoble-alpes.fr

***Abstract*—Formal methods for software development have made great strides in the last two decades, to the point that their application in safety-critical embedded software is an undeniable success. Their extension to non-critical software is one of the notable forthcoming challenges. For example, C programmers regularly use inline assembly for low-level optimizations and system primitives. This usually results in rendering state-of-the-art formal analyzers developed for C ineffective. We thus propose TInA, the first automated, generic, *verification-friendly* and trustworthy lifting technique turning inline assembly into semantically equivalent C code amenable to verification, in order to take advantage of existing C analyzers. Extensive experiments on real-world code (including GMP and ffmpeg) show the feasibility and benefits of TInA.**

***Index Terms*—Inline assembly, software verification, lifting, formal methods.**

I. Introduction

Context. Formal methods for the development of high-safety software have made tremendous progress over the last two decades [1], [2], [3], [4], [5], [6], with notable success in regulated safety-critical industrial areas such as avionics, railway or energy. Yet, the application of formal methods to more usual (non-regulated) software, for safety or security, currently remains a scientific challenge. In particular, extending the applicability from a world with strict coding guidelines and disciplined mandatory validation processes to more liberal and diverse development and coding practices is a difficult task.

Problem. We consider here the issue of analyzing "mixed code", focusing on the use of inline assembly in C/C++ code. This feature allows to embed assembly instructions in C/C++ programs. It is supported by major C/C++ compilers like GCC, clang or Visual Studio, and used quite regularly — usually for optimization or to access system-level features hidden by the host language. For example, we estimate that 11% of Debian packages written in C/C++ directly or indirectly depends on inline assembly, with chunks containing up to 500 instructions, while 28% of the top rated C projects on GitHub contains inline assembly according to Rigger et al. [7]. As a matter of fact, *inline assembly is a common engineering practice in key areas such as cryptography, multimedia or drivers. However, it is not supported by current state-of-the-art C/C++ program analyzers*, like KLEE [4] or Frama-C [1], possibly leading to incorrect or incomplete results. *This is a clear applicability issue for advanced code analysis techniques.*

Given that developing dedicated analyzers from scratch is too costly, the usual way of dealing with assembly chunks is to write either equivalent host code (e.g, C/C++) or equivalent logical specification when available. But *this task is handled manually* in both cases, precluding regular analysis of large code bases: manual translation is indeed time-consuming and error-prone. The bigger the assembly chunks are, the bigger these problems loom.

Goal and challenges. *We address the challenge of designing and developing an automated and generic lifting technique turning inline assembly into semantically equivalent C code amenable to verification.* The method should be:

Verification-friendly The produced code should allow *good enough* analyses in practice (informally dubbed *verifiability*), independently of the underlying analysis techniques (e.g., symbolic execution [8], [9], deductive verification [10], [11] or abstract interpretation [12]);

Widely applicable It should not be tied to a particular architecture, assembly dialect or compiler chain, and yet handle a significant subset of assembly chunks found in the wild;

Trustworthy The translation process should be insertable in a formal verification context without endangering soundness: as such it should maintain exactly all behaviors of the mixed code, and provide a way to show this property.

Verifiability alone is already challenging: indeed, straightforward lifting from assembly to C (keeping the untyped byte-level view) does not ensure it as standard C analyzers are not well equipped to deal with such low-level C code.

Scarce previous attempts do not fulfill all the objectives above. Vx86 [13] is tied to both the x86 architecture and deductive verification, while the recent work by Corteggiani et al. [14] focuses on symbolic execution. None of them addresses verifiability or trust. At first sight, decompilation techniques [15], [16], [17] may seem to fit the bill. Yet, as they mostly aim at helping reverse engineers, correctness is not their main concern. Actually, *"existing decompilers frequently produce decompilation that fails to achieve full functional equivalence with the original program"* [18]. Some recent works partially target this issue: Schwartz et al. [19] do not *demonstrate* correctness (they instead measure a certain degree of it via testing), while Schulte et al. [18] use a correct-by-design but intractable (possibly non-terminating) search-based method. Again, none of them study verifiability.

Breaking news 2
Code protection & obfuscation (ACSAC 2019)

BINSEC

By Mathilde Ollivier

- Problem = semantic attacks !
- Semantic attacks very effective
- Solution : path-oriented protections
- Strong, Cheap, Stealth
- « protection of VMx3, no cost »

Transformation (#TO/#Samples)	Dataset #1		Dataset #2		
	Goal 1 3h TO	*Goal 2 1h TO*	*Goal 1 24h TO*	*Goal 2 3h TO*	*Goal 2 8h TO*
Virt	0/46	0/15	0/7	0/7	0/7
Virt ×2	1/46	0/15	0/7	0/7	0/7
Virt ×3	5/46	2/15	1/7	0/7	0/7
SPLIT ($k = 10$)	1/46	0/15	0/7	0/7	0/7
SPLIT ($k = 13$)	4/46	0/15	1/7	1/7	0/7
SPLIT ($k = 17$)	18/46	2/15	3/7	2/7	1/7
FOR ($k = 1$)	2/46	0/15	0/7	0/7	0/7
FOR ($k = 3$)	30/46	8/15	3/7	2/7	1/7
FOR ($k = 5$)	46/46	15/15	7/7	7/7	7/7

How to Kill Symbolic Deobfuscation for Free
(or: Unleashing the Potential of Path-Oriented Protections)

Mathilde Ollivier
CEA, LIST,
Paris-Saclay, France
mathilde.ollivier2@cea.fr

Sébastien Bardin
CEA, LIST,
Paris-Saclay, France
sebastien.bardin@cea.fr

Richard Bonichon
CEA, LIST,
Paris-Saclay, France
richard.bonichon@cea.fr

Jean-Yves Marion
Université de Lorraine, CNRS, LORIA
Nancy, France
Jean-Yves.Marion@loria.fr

ABSTRACT

Code obfuscation is a major tool for protecting software intellectual property from attacks such as reverse engineering or code tampering. Yet, recently proposed (automated) attacks based on Dynamic Symbolic Execution (DSE) shows very promising results, hence threatening software integrity. Current defenses are not fully satisfactory, being either not efficient against symbolic reasoning, or affecting runtime performance too much, or being too easy to spot. We present and study a new class of anti-DSE protections coined as path-oriented protections targeting the weakest spot of DSE, namely path exploration. We propose a lightweight, efficient, resistant and analytically proved class of obfuscation algorithms designed to hinder DSE-based attacks. Extensive evaluation demonstrates that these approaches critically counter symbolic deobfuscation while yielding only a very slight overhead.

CCS CONCEPTS

• **Security and privacy** → **Software reverse engineering**; *Logic and verification*; Malware and its mitigation; • **Software and its engineering** → *Formal methods.*

KEYWORDS

Reverse Engineering; Code Protection; Obfuscation

ACM Reference Format:
Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. 2019. How to Kill Symbolic Deobfuscation for Free (or: Unleashing the Potential of Path-Oriented Protections) . In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3359789.3359812

ACSAC '19, December 9–13, 2019, San Juan, PR, USA

ACM ISBN 978-1-4503-7628-0/19/12…$15.00
https://doi.org/10.1145/3359789.3359812

1 INTRODUCTION

Context. Reverse engineering and code tampering are widely used to extract proprietary assets (e.g., algorithms or cryptographic keys) or bypass security checks from software. *Code protection* techniques precisely seek to prevent, or at least make difficult, such *man-at-the-end* attacks, where the attacker has total control of the environment running the software under attack. Obfuscation [21, 22] aims at hiding a program's behavior by transforming its executable code in such a way that the behavior is conserved but the program becomes much harder to understand.

Even though obfuscation techniques are quite resilient against basic automatic reverse engineering (including static attacks, e.g. disassembly, and dynamic attacks, e.g. monitoring), code analysis improves quickly [39]. Attacks based on *Dynamic Symbolic Execution* (DSE, a.k.a. *concolic execution*) [18, 30, 40] use logical formulas to represent input constraints along an execution path, and then automatically solve these constraints to discover new execution paths. DSE appears to be very efficient against existing obfuscations [5, 8, 13, 24, 37, 51], combining the best of dynamic and semantic analysis.

Problem. *The current state of symbolic deobfuscation is actually pretty unclear.* Dedicated protections have been proposed, mainly based on hard-to-solve predicates, like Mixed Boolean Arithmetic formulas (MBA) [52] or cryptographic hash functions [42]. Yet the effect of complexified constraints on automatic solvers is hard to predict [6], cryptographic hash functions may induce significant overhead and are amenable to key extraction attacks (possibly by DSE). On the other hand, DSE has been fruitfully applied on malware and legit codes protected by state-of-the-art tools and methods, including virtualization, self-modification, hashing or MBA [8, 37, 51]. A recent systematic experimental evaluation of symbolic deobfuscation [5] shows that most standard obfuscation techniques do not seriously impact DSE. Only nested virtualization seems to provide a good protection, assuming the defender is ready to pay a high cost in terms of runtime and code size [37].

Goals and Challenges. We want to propose a new class of dedicated anti-DSE obfuscation techniques to render automated attacks based on symbolic execution inefficient. These techniques should be *strong* – making DSE intractable in practice, and *lightweight* – with very low overhead in both code size and runtime performance. While most anti-DSE defenses try to break the symbolic reasoning