Black-box code analysis for reverse engineering through constraint acquisition and program synthesis

Grégoire Menguy

université
PARIS-SACLAY

Black-box code analysis for reverse engineering through constraint acquisition and program synthesis

Analyse de code en boîte noire pour la rétro-ingénierie via acquisition de contraintes et synthèse de code

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580, Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat : Informatique
Graduate School : Informatique et sciences du numérique. Référent : Faculté des sciences d'Orsay

Thèse préparée dans l'unité de recherche **Institut LIST (Université Paris-Saclay, CEA)**, sous la direction de **Julien SIGNOLES**, ingénieur-chercheur, le co-encadrement de **Sébastien BARDIN**, ingénieur-chercheur, le co-encadrement de **Nadjib LAZAAR**, maitre de conférence

Thèse soutenue à Paris-Saclay, le 14 mars 2023, par

Grégoire MENGUY

Composition du jury

Membres du jury avec voix délibérative

Ludovic Mé Professeur, Inria, Université de Rennes	Président
Valérie Viêt Triêm Tông Professeur, CentraleSupelec, Université de Rennes	Rapporteure & Examinatrice
Charlotte Truchet Maitre de conférence, Université de Nantes	Rapporteure & Examinatrice
Nathanaël Fijalkow Chargé de recherche, CNRS, Université de Bordeaux	Examinateur

THESE DE DOCTORAT

NNT : 2023UPASG023

Remerciements – Acknowledgments

Remerciements. Je tiens à remercier mes encadrants Julien Signoles, Sébastien Bardin, Nadjib Lazaar et enfin Arnaud Gotlieb pour leur soutien et leur disponibilité. Je remercie tout particulièrement Sébastien Bardin pour m'avoir initié au monde de la recherche et permis de réaliser des projets ambitieux et passionnants, Nadjib Lazaar pour son expertise en machine learning, Arnaud Gotlieb pour la rigueur apportée tout au long de la thèse, et enfin Julien Signoles pour ses conseils lors de la rédaction de ce manuscrit.

J'exprime ma gratitude à Valérie Viêt Triêm Tông, Charlotte Truchet, Ludovic Mé et Nathanaël Fijalkow pour avoir accepté de faire partie de mon jury.

J'adresse tous mes remerciements au CEA LIST et notamment au LSL pour m'avoir offert un cadre de travail motivant et convivial. Notamment, je tiens à remercier Lesly-Ann Daniel, Guillaume Girol et Olivier Nicole pour leurs retours constructifs sur mes articles, Adrien Gaonac'h pour nos discussions au bureau, qu'elles soient scientifiques ou non, et Frédérique Descreaux pour son efficacité au niveau administratif.

Enfin, je remercie ma famille, particulièrement mes parents qui m'ont accompagnés pendant toutes mes études, et ma compagne pour son soutien (notamment pendant le confinement), son écoute et ses retours constructifs pendant ces 3 années.

Pour conclure, je remercie tous ceux qui ont rendu cette thèse possible.

Acknowledgments. First, I thank my Ph.D. advisors: Julien Signoles, Sébastien Bardin, Nadjib Lazaar, and Arnaud Gotlieb. Particularly, I thank Sébastien Bardin, who introduced me to academic research and allowed me to work on ambitious and fascinating projects, Nadjib Lazaar for his machine-learning expertise, Arnaud Gotlieb for his rigor during my thesis, and finally, Julien Signoles for his advice.

I thank Valérie Viêt Triêm Tông, Charlotte Truchet, Ludovic Mé, and Nathanaël Fijalkow for being part of my jury.

Thanks to the CEA LIST, especially the LSL, for its excellent, motivating, and friendly atmosphere. Thanks to Lesly-Ann Daniel, Guillaume Girol, and Olivier Nicole for checking my papers before submission, to Adrien Gaonac'h for all our discussions, and to Frédérique Descreaux for her help in administrative procedures.

My thoughts go out to my family, particularly to my parents, who supported me during all my studies, and to my partner who helped me during the Ph.D. (especially during quarantine) and took the time to listen to my presentations, always giving helpful advice.

Finally, I thank anyone who made this Ph.D. possible.

Contents

Résumé long

Les logiciels deviennent de plus en plus grands et complexes. Ainsi, certaines tâches, pourtant cruciales, telles que le test et la vérification de code ou plus généralement la compréhension de code, sont de plus en plus difficiles à réaliser pour un humain. D'où la nécessité de développer des outils d'analyse de code automatiques.

L'analyse de code automatique permet de prouver des propriétés sur le code, comme la correction ou l'incorrection. Plus généralement, cela permet de mieux comprendre les programmes. Ces méthodes sont usuellement en boîte blanche, i.e., elles utilisent la syntaxe du code pour déduire ses propriétés via des raisonnements logiques. Ces méthodes sont très efficaces et ont été notamment utilisées par Microsoft, Facebook et Airbus. Néanmoins, elles présentent certaines limitations. Tout d'abord, elles nécessitent le code source qui n'est pas toujours accessible (ex: codes propriétaires, maliciels). De plus, la taille et la complexité des structures manipulées impactent drastiquement leur efficacité. Enfin, elles sont fortement impactées par la complexité syntaxique, pouvant être accentuée par des passes d'optimisation (améliorant la rapidité du code) et d'obfuscation (protégeant la propriété intellectuelle du code).

Cette thèse explore comment les méthodes en boîte noire peuvent inférer des propriétés utiles pour la rétro-ingénierie et la compréhension de code. Ces approches sont orthogonales aux approches en boîte blanche. Elles ne se reposent pas sur la syntaxe mais sur des exécutions du code et infèrent les propriétés voulues via des méthodes d'intelligence artificielle (IA) et notamment du machine learning (ML). Ainsi, elles n'ont pas besoin du code source mais uniquement du binaire. Elles peuvent analyser des programmes grands et complexes. De plus, elles ne sont pas impactées par la complexité syntaxique du code. Nous nous concentrons sur deux scénarios.

En premier lieu, nous étudions le problème de l'inférence de contrats de fonctions qui a pour objectif d'apprendre sur quelles entrées une fonction peut être exécutée pour obtenir les sorties souhaitées. Nous adaptons l'acquisition de contraintes, en résolvant une de ses principales limitations: la dépendance à un être humain. En est ressortie notre première contribution, PreCA, qui peut-être utilisée par des développeurs pour comprendre du code inconnu ou par des outils de vérification pour prouver la (in)correction du code. PreCA est la première approche totalement boîte noire et offrant des garanties de correction claires, le rendant particulièrement approprié pour l'aide au développement et la vérification.

Nous étudions ensuite le problème de la déobfuscation de code, qui vise à simplifier du code obfusqué. Notre seconde contribution, Xyntia, synthétise via des S-métaheuristiques une version compréhensible de blocs de code hautement obfusqués. Xyntia améliore grandement l'état de l'art. Il est plus robuste et plus rapide que les approches précédentes, qualités essentielles pour outrepasser le plus de protections possibles et être utilisable sur du code hautement obfusqué. De plus, nous proposons les deux premières protections efficaces contre la déobfuscation en boîte noire.

Tout au long de cette thèse, nous montrons le potentiel des méthodes en boîtes noire, liant l'IA et l'analyse de code. Nous pensons que ces travaux pourraient ouvrir de nouvelles directions de recherche pour l'analyse de code et l'IA. En effet, de nouveaux algorithmes d'IA pourraient être développés pour aider les analyses de code. Enfin, l'IA pourrait profiter de l'analyse de code, offrant un nouveau cas d'usage avec de nouvelles exigences et nécessitant ainsi de nouveaux compromis.

Abstract

Software always becomes larger and more complex, making crucial tasks such as code testing, verification, or code understanding more and more difficult for humans. Hence the need for tools to reason about code automatically.

Automated program analysis enables to prove code properties like correctness or incorrectness and more generally helps to understand software. Such methods are usually *white-box*, i.e., they rely on the code syntax to *deduce* code properties through *logical reasoning*. While white-box methods have proven to be very powerful, being used for example at Microsoft, Facebook, and Airbus, they also suffer from some limitations. First, they need the source code, which is not always available (e.g., proprietary software, malware). Second, the code size and the complexity of the data structures manipulated degrade their efficiency drastically. Third, they are highly impacted by syntactic code complexity, which optimizations (improving code speed and memory consumption) and obfuscation (impeding end-users from extracting intellectual property contained in the code) can amplify.

This thesis explores how *black-box* code analysis can *infer* valuable properties for reverse engineering and code understanding through *data-driven learning*. These approaches are completely orthogonal to white-box methods, as they do not use the code syntax but rely on executions to infer code properties using artificial intelligence (AI), especially machine learning (ML). As such, they do not need the source code (only the binary), can handle large and complex code, and are not impacted by syntactic code complexity. We focus, especially on two major application scenarios.

First, we consider the *function contracts* inference problem, which aims to infer over which inputs a code function can be executed to get good behaviors only. We adapt and extend the *constraint acquisition* learning framework, notably solving one of its major flaws: the dependency on a human user. It leads to our first contribution, PreCA, which can be used by developers to understand others' code like proprietary libraries or by automated program analyzers to verify software (in)correctness. PreCA is the first completely *black-box* approach enjoying clear theoretical guarantees. It makes PreCA especially suitable for development and verification uses.

Second, we consider the *deobfuscation* problem, which aims to simplify obfuscated code. Our second contribution, named Xyntia, synthesizes code block semantics through *S-metaheuristics* to offer a simple and understandable version of the code. Xyntia significantly improves the state-of-the-art, being a lot more robust and faster than prior work, which is crucial to bypass as many protections as possible and be usable over highly obfuscated codes. In addition, we propose the two first protections efficiently protecting against black-box deobfuscation.

Throughout this thesis, we show the potential of black-box methods, connecting AI to program analysis. We believe it opens the way for new research directions in the program analysis and AI communities. In the former community, new AI algorithms could be adapted to improve analyzers. In the later community, AI could benefit from the program analysis application scenario, which offers new requirements, hence new algorithm trade-offs.

Part I

Introduction

Chapter 1

Introduction

1.1 Context

Software is omnipresent in all areas from energy, transport, health, or defense. It brings many safety [1] and security [2] challenges, from software verification [3] to malware analysis [4]. It is crucial to handle them as security flaws can have huge economic [5] and geopolitical impacts [6].

Still, software is becoming always larger and more complex. In several areas, we observe exponential growth in software size [7]. Moreover, software usually relies on third-party, uncontrolled, and often not formally documented code. They can also combine different languages (e.g., 28% of the top C projects on GitHub contain inline assembly, according to Rigger et al. [8]) or integrate proprietary libraries whose source code is not available. All these new trends make crucial tasks such as code testing, verification, or code understanding more and more difficult for humans. For example, Vahabzadeh et al. [9] show that half of the 211 crawled projects from the Apache Software Foundation contains bugs in their test suite.

On top of this observation, software engineering pipelines regularly integrate optimization [10] or obfuscation [11, 12, 13] passes. Code optimization (resp., obfuscation) translates a program P to an equivalent but faster (resp., harder to understand) program P'. Such methods are usual in many contexts like in Android applications, where 50% of most popular apps are obfuscated according to Wermke and al. [14], web development [15], military code [16] and video games [17]. However, even if they conserve code *semantics*, they also impact code readability. For example, the function `clean` in Listing 1.1 can be obfuscated as presented in Listing 1.2. Here, `clean` and `obfuscated` are semantically equivalent, but `clean` is simpler to understand. Such transformations make manual program analysis even harder. Hence the need for robust tools to reason about code automatically and scale to big and complex code.

Program analysis. Automated program analysis enables to prove code properties like correctness, incorrectness, or more generally to help under-

```
1 int clean(int x, int y)
2 {
3     int ret = x-y;
4     return ret;
5 }
```

Listing 1.1: Clean toy example.

```
1 int obfuscated(int x, int y)
2 {
3     int ret = (x^-y)+2*(x&-y);
4     return ret;
5 }
```

Listing 1.2: Obfuscated toy example.

stand software. Different approaches exist like *abstract interpretation* [18, 19], *model checking* [20, 21] or *symbolic execution* [22, 23, 24, 25]. Such methods are *white-box*, i.e., they rely on the code syntax to *deduce* code properties through logical reasoning. While white-box methods have been proven to be very powerful, being used for example at Microsoft [26], Facebook [27], Amazon [28] and Airbus [29], they also suffer from some limitations. First, they need the source code, which is not always available (e.g., proprietary software, malware). Second, the code size and the complexity of manipulated data structures degrade their efficiency drastically. Third, they are highly impacted by syntactic code complexity, which optimizations and obfuscation can amplify. Such limitations can make these methods unusable. For example, trying to understand what `obfuscated` returns through symbolic execution [25, 30] would answer $(x \oplus -y)+2(x \wedge -y)$. This is not understandable by a human user and complementary simplification steps should be added to hopefully infer $x - y$. However, this can be even worst: analysis may never terminate or infer nothing [31, 32, 33].

Artificial intelligence. Meanwhile, *artificial intelligence (AI)* methods, in particular *machine learning*, have made strong progress enabling to solve very complex problems quickly. *Constraint programming* [34, 35] is a programming paradigm that only asks a developer to model problems, then is used by dedicated solvers to answer questions about the model. Constraint programming has notably been successfully integrated in many industrial processes [36, 37]. *Search algorithms* also brought strong success in many areas. In board games, AlphaGo [38] successfully beat the Go world champion Ke Jie in 2017. This was a strong achievement as Go has no clear heuristics to guide the algorithms. Another great success of AI is program synthesis, especially, programming by example, which helps a non-expert user to generate code automatically. A great achievement has been the integration of Flash Fill [39] in Microsoft Excel, which enables to synthesize functions manipulating strings.

1.2 Problematic

This thesis explores *how AI-based black-box code analysis can derive valuable properties for reverse engineering and code understanding through inference.* These are completely orthogonal to *white-box* (deductive) methods. Indeed,

they do not use the code syntax but collect a representative set of code behaviors by monitoring code executions. From such observed behaviors, *black-box* methods infer useful code properties using artificial intelligence (AI) and machine learning (ML). As such, they do not need the source code (only the binary) and can handle big and complex code. Moreover, they are not impacted by syntactic code complexity i.e., all code transformations that preserve code semantics do not impact *black-box* methods. Thus, *black-box* approaches are very promising to analyze highly optimized and obfuscated software.

Scenarios. We especially focus on two major application scenarios:

- *Function contract inference.* Contracts specify on which inputs a code function can be executed to get good behaviors only. Giving such contracts was shown to help code understanding and thus the full development process [40]. They are especially useful in contexts where the code is proprietary and thus not available at the source level. Contracts are not only useful for code understanding but also for automated program analyzers, like Frama-C [41], OpenJML [42], KeY [43], or Why3 [44], to verify software correctness. Our goal is to design a new *function contracts* inference method. It should be completely black-box to be usable on proprietary software, highly optimized code, or multiple languages contexts;

- *Deobfuscation.* In many contexts from military development [16] to video games [17], the software contains intellectual property that can be stolen by the end-users. As such, the software itself must be protected. To do so, the code is obfuscated, i.e., translated to an equivalent but more complex program. On the contrary, deobfuscation aims to simplify obfuscated code to help reverse engineering. Designing efficient deobfuscation methods is crucial. First, it enables to assess the robustness of obfuscation used by companies. Second, it helps fight against cybercrime. Especially, because it eases malware analysis. Our goal is to design a new *deobfuscation* method, completely *black-box* to be completely insensible to semantic preserving code transformations and so bypass state-of-the-art protections.

1.3 Challenges

Black-box code analysis enables to bypass many limitations of *white-box* analysis. It is insensible to syntactic code complexity and as such is not impacted by optimization or obfuscation passes. It is very general and enables to handle big, complex, and multi-language software. However, they also come with challenges that must be tackled to make them usable in practical applications.

First, we aim to analyze software whose source code is not available (malware, proprietary libraries). Thus, we cannot rely on prior user knowledge

as in our context, the code may be completely unknown to them. Especially, we cannot assume that the users can provide a representative set of inputs to exercise the code. Hence our methods should be fully automated.

Challenge 1 Analysis should be fully automated.

Second, in our black-box context, the only available information is observed input-output behaviors. Relying only on executions enables to be completely immune to syntactic code complexity introduced by obfuscation and optimization. Still, this is a major challenge to be able to infer useful properties from such sparse information.

Challenge 2 Few information should be needed by our analysis methods.

Third, proposed methods express properties in a given language. To be usable in practical contexts, from code understanding to deobfuscation, they must be robust i.e., able to infer a wide variety of properties of interest quickly. Of course, what "quick" means depends on the usage. For code understanding, the human-in-the-loop process is usual. Thus, inference methods must be fast enough to be used in such human-in-the-loop scenarios where the user leverages our methods to analyze local and complex code sections.

Challenge 3 Analysis must be robust and fast enough for human-in-the-loop scenarios.

Finally, the results of our black-box analysis are used to better understand code or verify software. As such, they must offer, if possible, clear correctness guarantees or, at least, return correct results with high probability.

Challenge 4 Inference method should enjoy clear correctness guarantees.

Of course, we would like to solve all these challenges at once. However, inferring useful properties in the *black-box* context is very difficult. Indeed, the search space is huge and cannot be adapted by observing the code syntax, which is not available. Just as *white-box* methods trade soundness or completeness for efficiency [18, 45, 22, 23], *black-box* methods must trade robustness or correctness for practical efficiency. Depending on the use case, we prefer to be more robust while in other cases clear correctness guarantees are crucial. Such considerations highly influence the choice of *inference* algorithms to use in order to extract information from code executions. This thesis focuses on two main scenarios: *function contract inference* and *deobfuscation*. The following describes their specific features and the challenges induced.

1.3.1 Function contract inference

Writing software is a tedious task needing a full understanding of programming languages, but also of libraries used. However, the available documentation is often not precise enough (if not even deprecated or missing). *Function contracts* enable solving these issues by specifying function behaviors formally.

For example, consider that a developer wonders over which inputs `clean` and `obfuscated` can be called to get a return value equal to 0. If we could design function contract inference tools, we would be able to infer that `clean` and `obfuscated` must be called with inputs s.t., $x = y$. Such a piece of information is really helpful for code understanding but it is not limited to it. Indeed, *function contracts* also help to prove software. This double usage (during the development and verification processes) asks for clear correctness guarantees. Indeed, if inferred *function contracts* are incorrect, developers could write unsafe and insecure code, while code verification could generate incorrect proofs. Thus, special care should be taken in **Challenge 4**.

1.3.2 Deobfuscation

The software contains intellectual property like secrets, cryptographic keys, or algorithms. Such valuable assets can be extracted by a malicious user. To impede reverse engineering the code, obfuscation methods are the only solution for now. Obfuscation will translate the clean version of the code to a very complex one that makes analysis very hard. For example, arithmetic expressions can be protected just like in Listing 1.2, useless code can be added or data (e.g., passwords, keys) can be encoded.

Thus, deobfuscation methods should be able to handle such excess of complexity. While the correctness of deobfuscation methods is important – the result will be used by a reverse engineer – it is less determining than for *function contract inference*. However, speed and robustness (**Challenge 3**) are crucial to be applied to many different codes and bypass a lot of protections. Otherwise, marginal protection changes could impede deobfuscation. In the *black-box* setting, the deobfuscator tries to infer the semantics of code blocks in an understandable format. Thus, being robust means handling a great variety of code quickly, not only simple ones.

1.4 Contributions

We devise new black-box AI-based code analysis methods to help reverse engineering and code understanding. They do not need the source code, but only the binary. Indeed, they deduce code properties by monitoring only code executions. Being *black-box*, they are not impacted by semantic preserving code transformations like state-of-the-art obfuscation and optimization methods.

Contributions. This thesis makes the following contributions:

- *We present different design choices adapted to each scenario.* As seen previously, distinct constraints can be induced by the application scenarios. This thesis demonstrates that *black-box* analysis can be applied to distinct scenarios leading to different strengths and limits. It shows that the choice of the inference algorithm is crucial. Especially, a balance should be found to propose methods adapted to an application and its constraints. Thereby, it relies on *active constraint acquisition* to infer function contracts with clear correctness guarantees. For deobfuscation, however, it uses *search-based* algorithms to focus on robustness;
- *We propose the first black-box function contract inference method based on constraint acquisition (Chapter 3).*
  - This is the first application of constraint acquisition (a learning framework from constraint programming) for program analysis. It offers a new application scenario with new specific constraints. Especially, it enables to remove the human user from the learning loop and replaces it with an oracle, solving thereby constraint acquisition main flaw.
  - We extend the usual constraint acquisition framework. We especially benefit from our application context (program analysis) to devise new extensions improving learning speed.
  - It leads to our new framework, named PreCA, the first black-box contract inference method with clear correctness guarantees. If PreCA language is expressive enough to represent the target contract, then we are sure to infer it;
- *We propose a new efficient black-box deobfuscation method (Chapter 4)*
  - We deepen the understanding of black-box deobfuscation. Especially, we dig into the previous MCTS-based black-box deobfuscator to assess and explain its strengths and weaknesses.
  - We propose Xyntia a new black-box deobfuscation method that solves previous approach flaws. We see the deobfuscation problem as an optimization one and advocate for the use of S-metaheuristics. Compared to prior works, Xyntia is a lot faster and more robust, i.e., able to deobfuscate semantically complex expressions. Xyntia also outperforms tested white-box approaches, based on rewriting rules, and grey-box ones, which combine white- and black-box views.
  - Finally, we propose the two first protections efficient against black-box deobfuscation. They locally increase code semantic complexity instead of syntactic one.

Throughout this thesis, we show the potential of black-box methods, connecting AI to program analysis. We believe it could pave the way for new research directions in program analysis and AI communities. In the former community, new AI algorithms could be adapted to improve analyzers. In the later community, AI could benefit from the program analysis application scenario, which offers new requirements, hence new algorithm trade-offs.

1.4.1 Secondary contributions

Publications. The work presented in Chapters 3 and 4 have been published in the following top tier conferences:

- *Search-Based Local Black-Box Deobfuscation: Understand, Improve and Mitigate*, Grégoire Menguy, Sébastien Bardin, Richard Bonichon and Cauim de Souza Lima. Published in the proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS).

- *Automated Program Analysis: Revisiting Precondition Inference through Constraint Acquisition*, Grégoire Menguy, Sébastien Bardin, Nadjib Lazaar, Arnaud Gotlieb. Published in the proceedings of the 31st International Joint Conference on Artificial Intelligence and the 25th European Conference on Artificial Intelligence (IJCAI-ECAI).

Code and artifacts. All code and artifacts have been released:

- The Xyntia framework is included in the open-source Binsec framework. Artifacts are available at `https://zenodo.org/record/5094898#.YO2UchMzbSw`.

- The PreCA framework is included in the open-source Conacq framework. Artifacts are available at `https://zenodo.org/record/6513522#.YnD1GnVfjmE`.

Research talks. Published works have also been presented in:

- International invitation based seminars:

  - **Schloss Dagstuhl – Leibniz Center for Informatics**
    Seminar entitled "Machine Learning and Logical Reasoning: The New Frontier", which aimed to foster discussions accross the machine learning and logic communities;

- International workshops:

  - **Machine Learning for Program Analysis (MLPA) 2020**
    Workshop co-localized with the IJCAI 2020 conference, which aims to encourage new discussions between the machine learning and program analysis communities;
  - **6th Franco-Japanese Cybersecurity Workshop**
    Workshop organized by Inria, which aims to help exchanges between French and Japanese researchers working on cybersecurity;
  - **KLEE workshop 2022**
    Workshop organized by the by the Software Reliability Group at Imperial College London and specialized on symbolic execution and related areas;

- National conferences:
  - **Conference on Artificial Intelligence and Defense (CAID) 2020 and 2021**
    Conference co-located with the European Cyber Week (ECW) and focused on the use of artificial intelligence for military applications;
  - **Forum International de la Cybersécurité (FIC) 2022**
    Europe's leading industrial forum on digital security and trust issues;

- National seminars:
  - **SoSySec seminar**
    Cybersecurity seminar co-organised by the Institut national de recherche en informatique et en automatique (Inria) and Direction Générale de l'Armement (DGA);
  - **2022 Annual Meeting of the WG "Formal Methods for Security"**
    Meeting organized by the Centre national de la recherche scientifique (CNRS) and focused on formal methods for security;
  - **Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information (RESSI) 2022**
    Meeting organized by the Centre national de la recherche scientifique (CNRS) and focused on security

- Internal seminars:
  - **Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM)**
    Seminar organised by the Coconut team from the LIRMM and specialized in artificial intelligence, especially constraint programming and constraint acquisition;

- **Simula Research Laboratory**
  Norwegian research institute;
- **Quarkslab**
  French company focused on software analysis for security;
- **Emproof**
  Company founded at the leading IT-security Horst-Görtz Institute for IT-Sicherheit (HGI) in Bochum and specialized on code obfuscation for embeded systems.

1.5 Outline

This thesis is organized as follows:

- Chapter 2 introduces the needed background to understand Chapters 3 and 4. It presents automated program analysis concepts from code semantics (Section 2.1) to symbolic execution and abstract interpretation (Section 2.2.1). Notably, it shows challenges induced by automatic static analysis and introduces dynamic analysis in Section 2.2.2. Section 2.3 presents the additional challenges of binary analysis and Section 2.3.3 shows that another layer of complexity can be added with obfuscation. Finally, Section 2.4 presents program and specification synthesis, which shapes the basis of both PreCA and Xyntia;
- Chapter 3 presents our first contribution: PreCA – for **Pre**condition **C**onstraint **A**cquisition. This is a completely black-box *weakest precondition* inference method, relying on *active constraint acquisition*. PreCA is the first black-box *weakest precondition* learner, showing good correctness guarantees. Indeed, state-of-the-art black-box methods ask the user for test cases or generate them stochastically. On the other hand, PreCA takes as input a language and generates test cases automatically. If its search language is expressive enough, PreCA is proved to infer the correct *weakest precondition*;
- Chapter 4 presents our second contribution: (i) the in-depth evaluation of the black-box deobfuscator Syntia; (ii) our new prototype Xyntia, outperforming Syntia in terms of speed and robustness; (iii) the first protections against black-box deobfuscation. Especially, we show why design choices of Syntia are not appropriate and why Xyntia is more efficient;
- Chapter 5 concludes the thesis. Section 5.1 resumes contributions presented in Chapters 3 and 4. It is followed (Section 5.2) by possible perspectives for the domain of black-box code analysis and more specifically for our two contributions PreCA and Xyntia.

Chapter 2

Background

We introduce in this chapter the background material necessary to understand and assess our contributions:

- Section 2.1 introduces program semantics, especially operational semantics. It enables to define properly what code execution means and is mandatory to formalize program analysis and synthesis;
- Section 2.2 gives an overview of automated program analysis methods, which aim to prove useful code properties. It presents both static and dynamic approaches and describes the pros and cons of each method;
- Section 2.3 presents when analyzing binary code is mandatory and the additional challenges of such analysis. Especially, it discusses two application scenarios: managed and adversarial binaries;
- Section 2.4 digresses from program analysis to give an overview of program and specification synthesis. Unlike program analysis, which aims to prove code properties, program synthesis aims to infer a code respecting a given specification. Closely related, specification synthesis takes a code snippet and infers its specification. Synthesis methods are strongly related to program analysis ones and are used all along this thesis;
- Section 2.5 presents how logic and constraints enable to define program properties and to reason about them. Logic is mainly used in automated proof systems while constraints are used to design models in constraint programming. It will present each framework, their relationships, and how they are used in program analysis.

2.1 Program semantics

To reason about program execution, it is necessary to formalize what is a program and what it executes. To do so, different formalisms have been

proposed like denotational and axiomatic semantics [46, 47, 48]. This thesis relies on the operational semantics [49], which describes an execution of a program as a sequence of simple steps.

First, let us formalize what a program is. In operational semantics, a program, often called a *command*, is run over a *memory state*. The combination of a command and a memory state is called a *configuration*, which represents the current memory state and the remaining commands to execute. As such, a command can either be a single command c or represent a full program as $c_1; c_2$ is also a command. The special command *skip* (which does not modify the memory state) is used to represent the computation end.

Definition 1. ***(Configuration)*** *A configuration is a tuple composed of a command c and a memory state s, denoted c/s.*

Configurations describe a program to run over a given memory state. We can then define the operational semantics, formalizing one execution step i.e., execution of one *command*.

Definition 2. ***(Operational semantics)*** *An operational semantics is a transition function "$\rightsquigarrow$" specifying transitions between pairs of configurations.*

We can then define the reflexive closure $\rightsquigarrow^*$, which corresponds to successive execution steps. This enables to describe full program execution and so, the following reachability properties: *end*, *diverge*, or *stuck*.

Definition 3. ***(End/diverge/stuck)*** *A configuration c/s:*

- *ends if and only if it exists s' s.t. $c/s \rightsquigarrow^* skip/s'$;*
- *diverges if and only if it can be derived infinitely ;*
- *is stuck if and only if there is no c'/s' s.t. $c/s \rightsquigarrow c'/s'$.*

Intuitively, they describe the possible behaviors of a code snippet. The "end" case means that the code was successfully executed (code terminated without crashing). The "diverge" case means the execution never terminates (i.e., infinite loop or infinite recursion). Finally, the "stuck" state enables the description of ***runtime errors*** (RTE). Especially, a configuration c/s leads to an RTE if and only if $c/s \rightsquigarrow^* c'/s'$ where c'/s' is stuck.

Example 1. *We describe here simple operational semantics for the toy language IMP from Table 2.1. IMP is a simple imperative language with conditionals (`if then else`) and loops (`while`). It manipulates integer values and enables to check internal states at run-time through the `assert` primitive. Each code construct is associated with semantics specifying code behavior. For example, when executing $(x := a)/s$ the first rule applies and updates the memory state s by setting the variable x to the value of the arithmetic expression a (noted $s[x \leftarrow [\![a]\!]]$). For the while loop, operational semantics provides two*

Table 2.1: IMP language syntax and evaluation rules

Grammar

$$
\begin{array}{rcl}
a & := & x \mid n \mid a_1 + a_2 \mid a_1 \times a_2 \mid a_1 \div a_2 \\
b & := & \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \neq a_2 \mid a_1 < a_2 \mid a_1 \leq a_2 \\
c & := & \text{skip} \mid \text{x := a} \mid c_1; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \mid \text{assert(b)}
\end{array}
$$

Evaluation rules

Commands:

$$\frac{}{(x := a)/s \rightsquigarrow skip/s[x \leftarrow [\![a]\!]]} \qquad \frac{}{(skip; c)/s \rightsquigarrow c/s} \qquad \frac{c_1/s \rightsquigarrow c_1'/s'}{(c_1; c_2)/s \rightsquigarrow (c_1'; c_2)/s'}$$

$$\frac{[\![b]\!]s = true}{(\text{if } b \text{ then } c_1 \text{ else } c_2)/s \rightsquigarrow c_1/s} \qquad \frac{[\![b]\!]s = false}{(\text{if } b \text{ then } c_1 \text{ else } c_2)/s \rightsquigarrow c_2/s}$$

$$\frac{[\![b]\!]s = true}{(\text{while } b \text{ do } c)/s \rightsquigarrow (c; \text{while } b \text{ do } c)/s} \qquad \frac{[\![b]\!]s = false}{(\text{while } b \text{ do } c)/s \rightsquigarrow skip/s}$$

$$\frac{[\![b]\!]s = true}{assert(b)/s \rightsquigarrow skip/s}$$

Boolean expressions:

$$\frac{a_1/s \rightsquigarrow a_1'/s}{(a_1 \bullet a_2)/s \rightsquigarrow (a_1' \bullet a_2)/s} \qquad \frac{a_2/s \rightsquigarrow a_2'/s}{(n \bullet a_2)/s \rightsquigarrow (n \bullet a_2')/s} \qquad \frac{n \bullet m}{(n \bullet m)/s \rightsquigarrow true/s}$$

$$\frac{\neg(n \bullet m)}{(n \bullet m)/s \rightsquigarrow false/s}$$

Arithmetic expressions:

$$\frac{[\![x]\!]s = n}{x/s \rightsquigarrow n/s} \qquad \frac{a_1/s \rightsquigarrow a_1'/s}{(a_1 \diamond a_2)/s \rightsquigarrow (a_1' \diamond a_2)/s} \qquad \frac{a_2/s \rightsquigarrow a_2'/s}{(n \diamond a_2)/s \rightsquigarrow (n \diamond a_2')/s}$$

$$\frac{r = n \diamond m \qquad \diamond \neq \div}{(n \diamond m)/s \rightsquigarrow r/s} \qquad \frac{m \neq 0 \qquad r = n \div m}{(n \div m)/s \rightsquigarrow r/s}$$

where x represents a variable, c a command, a an arithmetic expression, b a Boolean expression, n, m, r constant values, $\bullet \in \{=, \neq, <, \leq\}$, $\diamond \in \{+, \times, \div\}$ and $[\![a]\!]$ (resp. $[\![b]\!]$) is the evaluation of the arithmetic expression a (resp. Boolean expression b).

rules, one when the condition is true and another when it is false. Observe that there is no rule in the operational semantics to evaluate a division by 0 *or an* `assert(false)`*. This is used to describe stuck states.*

From this language and operational semantics, we can define 3 programs describing each code behavior:

1. ***end.*** $(x := 0; (\ \textit{while}\ x < 10\ \textit{do}\ x := x + 1); \textit{skip})/s \rightsquigarrow^* \textit{skip}/s[x \leftarrow 10]$
2. ***diverge.*** $(x := 0; (\ \textit{while}\ \textit{true}\ \textit{do}\ x := x + 1); \textit{skip})/s \rightsquigarrow^*$
3. ***stuck.*** $(x := 0; \textit{assert}(x \neq 0); \textit{skip})/s \rightsquigarrow^* (\textit{assert}(x \neq 0); \textit{skip})/s[x \leftarrow\ 0]$

Note

While giving the programming language grammar along with its operational semantics is necessary for white-box methods (Section 2.2.1), in the previous example, we only give them for clarity. Indeed, in our contributions (Chapters 3 and 4), we propose *black-box* methods. These observe code behaviors and infer properties based on them. As such, in our context, operational semantics is used to define precisely what are possible code behaviors: *end*, *diverge*, or *stuck*.

2.2 Automated program analysis

Automated program analysis enables to prove software properties. In general, it aims to prove reachability properties, i.e., given a command c, is there an initial memory state s s.t., $c/s \rightsquigarrow^* c'/s'$ where the state c'/s' is of interest (e.g., it is an error state). A wide variety of approaches exist. Static methods read the code syntax to deduce properties from it. They require a white-box access to the code (source code or binary). Dynamic methods extract information from observed code executions. They can be white-box (reading execution traces) or black-box (relying only on input-output behaviors). Ideally, such analyzers would be fully automated, always terminate, be sound (no false positive, i.e., never returns that a property hold if it is not) and complete (no false negative, i.e., never returns that a property does not hold if it is). However, Rice's theorem [50] states that *any semantic property of a language recognized by a Turing machine is undecidable*. For example, there is no decision procedure $\mathcal{P}$, such that, for all programs P, $\mathcal{P}$ states in finite time if P never raises an error. From a reverse engineering point of view, there is no decision procedure $\mathcal{P}$, such that, for all pairs of programs (P_1, P_2), $\mathcal{P}$ can state in finite time if P_1 and P_2 are semantically equivalent [50].

Such an impossibility result led to the creation of analysis methods that give up on properties to work in practice. Static analysis methods reasons on the code syntax to deduce its properties. They can over-approximate or under-approximate the possible code behaviors. In the former case, methods

focus on soundness and renounce completeness. In the latter one, the analysis aims for completeness but loses soundness. Another approach, called dynamic analysis, proposes to execute the software to infer useful properties. In such a case, the analysis is complete but not sound.

The following proposes a brief presentation of major static and dynamic approaches. Concerning static methods, we present the difference between over- and under-approximation-based analysis. We especially touch on abstract interpretation, symbolic execution, and model checking as they are used in many code understanding, reverse engineering, and deobfuscation frameworks. Weakest precondition calculus is presented extensively in Chapter 3, which especially tackles this problem. Concerning dynamic methods, which are by nature under-approximation-based, we describe the difference between the white- and the black-box framework that is used all along the thesis.

2.2.1 Static program analysis

Static program analysis reads the code – be it high levels like JAVA and Python or low levels like C or assembly – to infer properties on it. These are *white-box methods* that can be used to prove software correctness [18, 45, 51, 52, 53, 54], find bugs [55, 25, 56, 57], or reverse engineer software [58, 59, 30, 60, 61, 62]. As stated previously, all static program analysis methods must renounce some good properties to work in practice. Here we present approaches making different choices to match their respective use-case.

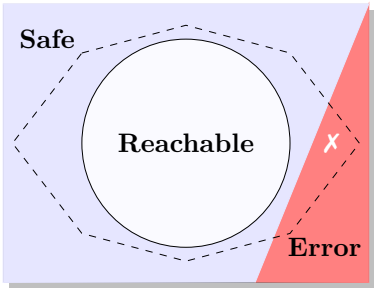


Figure 2.1: Over-approximation

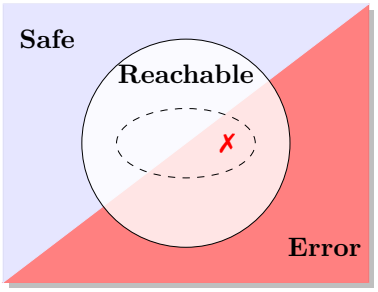


Figure 2.2: Under-approximation

2.2.1.1 Over-approximation-based analysis

Static program analysis can over-approximate the possible behaviors of the code, e.g., abstract interpretation [18, 19]. Fig. 2.1 describes such a setting: all reachable code behaviors are included in the approximation (represented by the dotted polygon). Thus, over-approximation-based methods are sound. It makes them especially appropriate to prove software automatically. However, completeness (i.e., all found bugs are real bugs) is not prominent, and to work in practice, these methods renounce to it. Thus, it may return false alarms,

i.e., unreachable bugs. For example, in Fig. 2.1, the over-approximation of considered behaviors includes error states which are in fact unreachable. Still, if it raises no alarm it means that the code under analysis is correct.

The false alarm problem. Over-approximations of code behaviors introduce imprecisions. If such imprecisions are too high, many false alarms, warning for errors that are in fact not reachable, are raised. Finding a good balance between precision and efficiency is thus necessary to verify complex code. However, it is also a very hard task needing for now human experts. In Fig. 2.1, the analysis is too imprecise and is not able to prove that the code is safe.

2.2.1.2 Under-approximation-based analysis

Other methods, like *symbolic execution* [22, 23, 24, 25] and *bounded model checking* [21, 63], under-approximate the possible behaviors of the code. As presented in Fig. 2.2, only reachable code behaviors are included in the approximation (represented by the dotted ellipse). Still, some reachable ones are not included. Thus, these methods renounce soundness to keep completeness. They are fully automatized methods that can prove error existence. However, if no error is found, it does not mean there is no error. Indeed, being unsound, they can miss some code behaviors. For example, in Fig. 2.2, the approximation does not see some error states which are in fact reachable.

Path explosion problem. Software can present many or even an infinite number of different execution paths. However, under-approximation-based analysis can only consider a finite set of execution paths. This causes the path explosion problem: so many paths exist that analysis will not be able to handle all of them. For example, symbolic execution exercises one path at a time. Thus, it needs to prioritize promising paths first using ad-hoc heuristics or machine learning [64]. On the other hand, bounded model checking extracts the formula representing the code for paths of size up to a fixed bound.

Remark

In general, under-approximation-based analysis cannot prove software. Still, if the number of paths in the code is finite, it can prove code correctness by exercising all paths. In such a case, it is sound and complete – but is restricted to special programs. Some approaches rely on such methods to prove code correctness [65].

2.2.1.3 Limitations

Static methods allow to prove properties on code. While highly powerful, some practical limitations arise. First, they are highly sensitive to syntactic code complexity introduced by obfuscation and optimization passes. Second, as the code is not really executed, it is unclear how to handle system calls

(not always well specified). The usual method consists in writing stubs for system calls and dynamic libraries functions, which is time-consuming and error-prone. Third, as we have seen, they suffer precision or path explosion problems making them irrelevant in some practical contexts.

Link to the thesis

This thesis studies application scenarios where static analysis struggles and is not usable. We propose two new frameworks, based on black-box dynamic analysis, outperforming static ones.

2.2.2 Dynamic program analysis

An orthogonal solution to static analysis is *dynamic analysis* – also named data-driven analysis. It runs the code over concrete inputs to observe how it behaves. Because it executes the code, no problem with system calls or library calls arises: they are executed. Moreover, even if such methods are not sound (the code is run over a finite set of inputs), it is usually complete. The following discusses the difference between *black-* and *white-box* dynamic analysis.

2.2.2.1 Black-box dynamic analysis

In black-box dynamic analysis, the code is simply run over a finite set of inputs. The analyzer monitors observable input-output (I/O) behaviors like standard outputs, network connections, system calls, library calls, and returned values. However, in the black-box context, the code is supposed to be a black-box and execution traces cannot be extracted. In terms of operational semantics, black-box methods monitor a full execution $c/s \rightsquigarrow^* skip/s'$ without any access to the atomic $\rightsquigarrow$ operator.

In the literature, dynamic methods can analyze the full code or focus on local sub-parts. It depends on the use-case. For example, in fully automated methods, analyzers execute more likely the full code over concrete inputs. It enables to observe the behavior(s) of the full software. This is especially useful to perform behavioral malware analysis [4] or black-box fuzzing [66]. However, in man-in-the-loop scenarios where analyses help reverse engineering the code, analyses can focus on a small piece of code like a code block or a function. This enables to observe simpler behaviors that can be more easily understood. Such local analyses are especially used in deobfuscation [67] or precondition inference [68, 69, 70].

2.2.2.2 White-box dynamic analysis

White-box dynamic methods extend black-box ones by not only monitoring observable behaviors but also execution traces. In terms of operational seman-

tics, it monitors all or parts of the execution steps $c/s \rightsquigarrow c'/s'$ – note that we use $\rightsquigarrow$ and not its transitive closure $\rightsquigarrow^*$. All these data can be highly useful in different contexts, e.g., to handle self-modifying code [71] or to recover part of the control-flow graph [72]. Especially, white-box dynamic analysis is used in behavioral malware analysis [4], deobfuscation [73], grey-box fuzzing [74], and invariant inference [40].

2.2.2.3 Limitations

Independently of the approach used, dynamic methods come with inherent limitations. First, the choice of the inputs to exercise the code is prevalent. Wrongly chosen inputs may lead to useless observations. Especially, in some contexts like fuzzing, code coverage is highly important, and finding a good set of inputs is crucial [75]. Second, in local scenarios, finding good *reverse windows* – i.e., a sub-part of the code to focus on – is highly important. A wrongly chosen *reverse window* could lead to overly complex behaviors impeding analysis [67, 76, 73]. Third, dynamic analysis needs a compiled version of the code. As such, observed behaviors could be architecture dependant or could depend on the compiler and compiling optimizations used. Finally, some errors can remain silent like stack buffer overflows making the detection of incorrect behaviors harder.

Link to the thesis

This thesis proposes two *black-box* analysis methods. They enable bypassing presented limitations of *white-box* methods relying on *abstract interpretation*, *symbolic execution*, or *white-box dynamic analysis*. To do so, they combine *black-box* dynamic analysis with *artificial intelligence*.

2.3 Binary-level code analysis

Our work can be beneficially used to analyze binary code i.e., compiled version of the software. The following presents the different applications where binary-level code analysis is needed and what are its main challenges.

2.3.1 Application scenarios

Binary analysis is needed when the source code of the software is not available. This is the case in different contexts like:

1. *Malware.* Malware authors often release malware in binary format. Indeed, binary analysis is known to be harder than source-level analysis. Thus, it enables to impede detection and reverse engineering;

2. *Third-party code.* Companies who want to share or sell software usually release binary executables. This helps to protect intellectual property by integrating for example water-marking – which could be removed during the compilation process otherwise. Such water-marking can then be used to detect fraudulent use/sharing of the software [77];

3. *Legacy code.* It is not rare that companies use obsolete code whose source is lost. As some crucial services may need this code, it is important to understand the binary, in order to use it well, or try to refactor the code.

Independently of the context, working at the binary level impedes the use of efficient source-level analyses and makes hand-made reverse engineering a lot harder. Moreover, by manipulating the binary code, authors can integrate very complex obfuscation methods, breaking abstractions kept by the compiler.

2.3.2 Challenges of managed binary analysis

Analyzing managed binaries, i.e., obtained by compiling clean source codes through legit compilation workflows, comes with additional complexity compared to source code analysis. Especially, binary-level analysis can rely on fewer abstractions compared to source-level analysis.

- *Stripped binaries.* Much information is lost during compilation. Especially, if the binary is stripped, all function and variable names are removed. This makes reverse engineering by a human tedious. Thus, methods propose to recover such debug information [78, 79, 80];

- *Unknown control-flow graph (CFG).* At the binary level, retrieving the control-flow (`if`, `else`, `switch`, `for`, `while`) is undecidable. This makes automated binary analysis a lot harder as many methods need the CFG to work. As such, different disassembly methods have been proposed like linear and recursive disassembly. However, they all have strengths and drawbacks;

- *Untyped memory.* Assembly code directly operates on a finite set of registers and a unique, large, untyped memory. The next instruction to execute is referenced by the program counter (`eip` in x86). A memory area, called the stack is used to keep track of function local variables. In x86, the stack is delimited by the registers `ebp` and `esp`. Thus, function arguments and variables are referenced as offsets of these registers and any addresses is reachable. It makes analysis a lot harder.

2.3.3 Challenges of adversarial binary analysis

On top of the previously presented challenges, another complexity layer can be added on purpose to protect valuable assets (e.g., encryption keys and pro-

prietary algorithms) contained in the software. Such adversarial binaries rely on *obfuscation* [77, 11] to protect code against the user, who can read, run and tamper with the released (binary) code. While provably secure obfuscation is impossible in such a context [81], efficient anti-reverse-engineering methods can be included in the released version of the code to make analyses a lot harder (more expensive in time or in money).

Obfuscation [77, 11, 82] is, thus, a set of methods designed to make reverse engineering (understanding program internals) hard. It transforms a program P to a functionally equivalent, more complex program P' with an acceptable performance penalty. Obfuscation aims to delay the analysis as much as possible in order to make it unprofitable. Thus, it is especially important to protect the software from *automated program analysis* and different methods have been proposed to kill previously presented *white-box* analysis methods [32, 83].

Remark

Obfuscation distinguishes from so-called *security by obscurity*. In the latter case, security is enforced by the fact that the (crypt-)analyst does not know which algorithm is used. In obfuscation, the protection algorithm is known but the seed to initialize it is the secret. Thus, it must rely on hard program analysis problems. This way, even if the reverse engineer knows which obfuscation is used, analyzing the code remains very hard.

Link to the thesis

Methods proposed in this thesis are especially appropriate to handle binary and obfuscated code. They need few pieces of information, which is pleasant to handle the loss of abstraction at the binary level. Moreover, being black-box, they are insensitive to obfuscation. In Chapter 4, we especially devise a deobfuscator, simplifying highly obfuscated code blocks.

2.4 Program synthesis

While verification takes a program and a specification to prove code correctness, *program synthesis* and *specification synthesis* explore different directions. Especially, *program synthesis* takes a specification and generates a program from it. Conversely, *specification synthesis* takes a program and infers its specification. Program and specification synthesis are closely related. The following presents and discusses each approach.

2.4.1 Program synthesis

Program synthesis aims to infer a code implementation based on a user-given specification. Such a specification can be formal [84, 85], given in natural language [86, 87, 88] or given through input-output (I/O) examples [89, 90, 67]. Historically, program synthesis is viewed as a problem from deductive theorem proving [84]. It takes a formal specification (P, R), where P is a predicate over the function inputs and R is a predicate over function inputs and outputs. It computes from it a constructive proof for the sentence $\forall a, P(a) \implies \exists z, R(a, z)$, hence generating a program coherent with the specification. In such a context, program synthesizers take as input a formal specification and a set of deduction and re-writing rules.

Syntax-guided synthesis. Depending on the application scenario considered (synthesizing SQL queries, web application, or Excel MACRO) it may be beneficial to enforce a dedicated *domain-specific language (DSL)*. Syntax-guided synthesis enables to restrict the search-space by specifying such DSL. The induced search space can then be explored through enumerative search [91, 92, 93], constraint solving [94] or stochastic search [10]. It also enables to devise appropriate heuristics, hence improving efficiency.

Programming by example (PBE). Synthesis methods fed with formal specifications need experts to specify the desired behaviors. *Programming by example* proposes to replace such a formal specifications by user-given input-output examples. This is a big drive in program synthesis as it enables non-expert users to synthesize code. For example, PBE has been successfully integrated into Microsoft Excel [39]. If the examples are given by the user, the method is said to be *passive*. On the other hand, if the examples are automatically generated, then the method is said to be *active*.

2.4.2 Specification synthesis

Unlike program synthesis, which takes a code specification and infers an implementation for it, *specification synthesis* [40, 95, 68, 96, 69] takes a code implementation and tries to infer its specification i.e., on which inputs the code is correct. Program synthesis and specification synthesis are highly related. Especially, it is often useful to define DSL to describe specifications, paving the way for syntax-guided methods [70, 68]. Moreover, code behaviors can also be approximated by a finite set of examples. The following discusses two approaches: white- and black-box synthesis.

White-box specification synthesis. White-box methods take as input a formal representation of the code to synthesize its specification. It often relies on static code analysis to infer the specification or prove its correctness. A popular framework is Counter-Example Guided Inductive Synthesis (CEGIS). Usually applied to program synthesis, it is also commonly used in specification

synthesis [97, 98, 99, 100]. It generates candidate solutions and queries a program analyzer to check their correctness. If the solutions are incorrect, they are refined thanks to the counter-examples given by the static analyzer.

Black-box specification synthesis. Still, extracting code semantics to synthesize its specification can be very challenging (obfuscation, multi-language code, etc). Thus, just like PBE relying on I/O examples, *black-box* approaches approximate the code behaviors with I/O examples. Some methods [40] only rely on positive examples – i.e., good executions of the code – while others combine positive and negative ones for improved precision [70, 68]. Moreover, such methods can be passive or active.

2.4.3 Limitations

Both program and specification syntheses come with limitations. Especially, programming by example and black-box specification synthesizers must take as input a representative set of I/O examples. How to generate this set is still a hot topic. Some approaches consider that the user is able to give such examples [40], while others do not [68]. In the former case, passive approaches (needing user examples) can be applied. In the latter case, synthesizers must generate examples themselves. Moreover, as examples only approximate the semantics of the code, they do not enjoy clear correctness guarantees. Improving confidence in synthesized results is an important problem.

Link to the thesis

This thesis devises two new *black-box* analysis methods using synthesis to extract properties from code executions. Chapter 3 proposes an *active black-box specification synthesizer* to infer with clear guarantees function contracts. Chapter 4 proposes a new *passive* and *stochastic programming by example* method for *black-box deobfuscation*.

2.5 Logic and constraints to specify code

Over the last decade, more and more program analysis and synthesis methods rely on logic [20, 22]. It enables to specify formally code, its properties, and to reason about them. Such formal reasonings also expresses deduction rules to prove code properties. An alternative to logic is *constraints* [101] from the *constraint programming* community. The following presents fundamental definitions of logic and constraints and how they have been used for program analysis.

2.5.1 Logic

Logic is composed of a *language* and a *semantics*. The language is described using a syntax defining well-formed formulas. On the other hand, the semantics maps code constructs to values.

Propositional logic. The simplest standard logic is the *propositional logic* representing propositions, i.e., statements that can be true or false. The language associated describes all well-formed formulas with *propositional connective symbols* ($\neg, \wedge, \vee, \implies, \iff$). Each symbol is Boolean. Each well-formed formula is associated to a semantics, i.e., a function that assigns meaning to it. For the *propositional logic*, the semantics is recursively defined thanks to the usual truth tables for *propositional connective symbols*. An example of a propositional logic formula is: $[(x \implies y) \wedge x] \implies y$, where x, y, and z are Boolean variables.

First order logic. First-order logic extends *propositional logic* with quantifiers ($\exists$ and $\forall$) and non-logical symbols. Restrictions of a logic are called a fragment. Especially, *quantifier-free first-order logic* is a fragment of *first-order logic* with no quantifier. The language extends the *propositional logic* language with a finite number of predicate and function symbols over the non-logical symbols. Thus, first-order logic formulas enable to describe relations between elements. The semantics inherits the meaning of propositional connective symbols ($\wedge, \vee$, etc) from the *propositional semantics*. Predicate and function symbols can be given any meaning depending on the usage. Still, first-order logic is often combined with theories, e.g., the linear integer arithmetic (LIA) theory. Such a theory defines the semantics of predicate and function symbols. Moreover, it extends the semantics with axioms from the underlying theory. For example, the LIA theory manipulates 0-ary function symbols representing integers. Predicates to reason about integers are also added, e.g., the "$\star = \star$" states that two integers are equal. On top of it, the theory adds new axioms, like the reflexivity or the transitivity properties of "$\star = \star$". An example of a *first-order logic* formula for LIA theory is: $\forall x, \exists (y, z), x = y + z$ where x, y and z are integers. Here, the theory defines the meaning of the "$\star = \star$" predicate and the 2-ary function symbol "$\star + \star$" over integers.

Solvers. Decision procedures [102] enable solving formulas expressed in some logic. Depending on the logic considered, different algorithms may exist. SAT solvers are decision procedures dedicated to solve propositional formulas. They often rely on Conflict-Driven Clause Learning (CDCL) [103]. Satisfiability Modulo Theories (SMT) solvers extend SAT solvers to handle first-order logic. They often rely on DPLL(T) [104], an extension of the Davis-Putnam-Loveland-Logemann (DPLL) algorithm for SAT solving, intertwining SAT with reasoning over some theory T.

Logic for program analysis. Logic, and especially first-order logic, is often used to express code properties. It can be used to express formally the derived specification of a code snippet [51], the behaviors of code [21], of an execution path [105, 25], or of a set of execution paths (hyper-properties) [65, 106, 107]. Combined with solvers, and more generally to proof systems, it has been used to prove useful code properties like memory safety [108, 51], noninterference [109] or incorrectness [57].

Link to the thesis

In Chapter 4, we create a new program synthesizer for *black-box deobfuscation*. It takes as input a quantifier-free first-order logic for the BitVector theory. The language defines our search space and the theory defines predicates and function symbols' meanings.

2.5.2 Constraints

Constraints [101, 110] are used in *constraint programming* to model problems and reason about them. Constraints are written in a language made of constant values (0-ary function symbols in *first-order logic*), functions (n-ary function symbols with $n > 0$ in *first-order logic*), and constraint relations (predicate symbols in *first-order logic*).

Constraint domain. A constraint C is defined by its domain, which specifies:

- Its syntax, that is the constant values, functions, and constraint relations used. Moreover, it describes the number of arguments of the constraints and how to place them. For example, consider the $C(X, Y) : X + Y = 0$ constraint. It has two arguments. The first (resp. second) one must be placed on the left (resp. right) of the "+" function, which takes two arguments. It also contains a constraint relation "=" and the constant value "0". Constraints can also take a non-fixed number of variables. In such a case, they are called *global constraints*;

- The values the variables can take. For example, the $C(X, Y) : X+Y = 0$ constraint can have different meanings depending on whether X and Y are integers or reals;

- The meaning of each symbol i.e., the result of applying the constraint on its arguments. For example, to describe $C(X, Y) : X + Y = 0$, it must define the meaning (i.e., the semantics in *first-order logic*) of the "+" function, the "=" constraint relation and the "0" constant value.

Standard restrictions. In general, constraint programming is used to solve combinatorial problems and variables usually take a finite number of values.

Moreover, the constraint language includes no quantifiers and only the conjunctive logical connector. Thus, a constraint is a conjunction of atomic constraints.

Solvers. Constraint programming solvers usually rely on domain and interval consistency, and backtracking [111, 112] to solve combinatorial problems. Also, a cost function can be added to solve optimization problems [110]. Links between static analysis and constraint programming can also be highlighted. Especially, Pelleau et al. [113] rely on abstract interpretation to devise a new solver.

Constraints for program analysis. Constraints have been used to encode software semantics. It usually requires to translate code under analysis to *static single assignment* (SSA) form. In such a form, each code statement can be translated into a constraint over basic types (e.g., integers) or more advanced ones (e.g., arrays). A basic block is then the conjunction of each constraint associated with block statements. Some code statements (e.g., variables assignment) are translated to atomic constraints, while others (e.g., if then else, while loops) are translated to global constraints. From such an encoding, constraint programming solvers can then be leveraged to show functional properties, generate counter-examples, or generate test-cases automatically [114, 115].

Link to the thesis

In Chapter 3, we create a new function contract inference method. It relies on *constraint acquisition*, a learning framework enabling to infer constraints to model a problem. As such, our method is limited to infer conjunctions of atomic constraints. We show how to bypass this limitation in Section 3.5.

Part II

Contributions

Chapter 3

Synthesizing function contracts

Abstract

Program annotations under the form of function pre/postconditions are crucial for many software engineering and program verification applications. Unfortunately, such annotations are rarely available and must be retrofit by hand. In this paper, we explore how Constraint Acquisition (CA), a learning framework from Constraint Programming, can be leveraged to automatically infer program preconditions in a *black-box manner*, from input-output observations. We propose PreCA, the first ever framework based on active constraint acquisition dedicated to infer memory-related preconditions. PreCA overpasses prior techniques based on program analysis and formal methods, offering well-identified guarantees and returning more precise results in practice.

3.1 Introduction

Program annotations under the form of function pre/postconditions [52, 54, 53] are crucial for the development of correct-by-construction systems [116, 117] or program refactoring [40]. They can benefit both a human or an automated program analyzer, typically in software verification where they enable scalable (modular) analysis [108, 118]. Unfortunately, annotations are rarely available and must be retrofit by hand into the code, limiting their interest – especially for black-box third-party components.

Problem Efforts have been devoted to *automatically infer* preconditions from the code, and contract inference is now a hot topic in Program Analysis and Formal Methods [119, 40, 70, 120, 68]. Since this problem is undecidable (as most program analysis problems), the goal is to design principled methods with good practical results.

Yet, the state-of-the-art is still not satisfactory. While *white-box approaches* leveraging standard static analysis [52, 54, 53, 119] can be helpful, they quickly suffer from precision or scalability issues, have a hard time dealing with complex programming features (loops, recursion, dynamic memory) and cannot cope with black-box components. On the other hand *black-box methods*, leveraging test cases to dynamically infer (likely) function contracts [40, 70, 68], overcome static analysis limitations on complex codes and have drawn attention from the software engineering community [121]. Yet, they heavily depend on the quality of the underlying test cases, which are often simply generated at random, given by the users [40] (passive learning), or automatically generated during the learning process – but without any clear coupling between sampling and learning [70, 68] – and so, show no clear guarantee on the inference process.

Constraint Acquisition. Constraint programming (CP) [101] has made considerable progress over the last forty years, becoming a powerful paradigm for modelling and solving combinatorial problems. However, modelling a problem as a constraint network still remains a challenging task that requires expertise in the field. Several constraint acquisition (CA) systems have been introduced to support the uptake of constraint technology by non-experts.

Especially, rooted in version space learning, Conacq is presented in its passive and active versions [122]. Based on solutions and non-solutions labelled by the user (acting as an oracle), the system learns a set of constraints that correctly classifies all examples given so far. This is an active field of research, with many proposed extensions, for example allowing partial queries [123].

However, even though Conacq enjoys strong theoretical foundations, such CA systems are hard to put in practice, as they require to submit *thousands of queries to a user*. In automated program analysis, the huge number of queries is not a problem as long as a program plays the oracle.

Goal and contributions. In this paper, we explore the potential of Constraint Acquisition for *black-box precondition inference*. To the best of our knowledge, this is the *first* application of CA to program analysis and our overall results show its potential there. Our main contributions are the following:

- We propose PreCA, the first ever (Conacq-like) framework based on active constraint acquisition and dedicated to infer preconditions (Section 3.4). We show in Section 3.4.3 that PreCA enjoys much better theoretical correctness properties than prior black-box approaches. Indeed, if our learning language is expressive enough, PreCA is *guaranteed* to infer the *weakest precondition*;

- We describe a specialization of PreCA to the important case of memory-related preconditions (Section 3.5). Especially, we propose a dedicated constraint language (including memory constraints) for the problem at

hand, as well as domain-based strategies to make the approach more efficient in practice (Section 3.5.2);

- We experimentally evaluate the benefits of our technique on several benchmark functions (Section 3.6.1). The results show that PreCA significantly outperforms prior precondition learners, be it black-boxes or white-boxes – which came as a surprise. For example, PreCA with 5s budget per sample performs better than prior approaches with 1h per sample.

Overall, it turns out that seeing the precondition inference problem as a Constraint Acquisition task is beneficial, leading to good theoretical properties and beating prior techniques.

3.2 Background

3.2.1 Preconditions and Weakest Preconditions

Notations

This contribution focuses on inferring function precondition. For clarity, we add syntactic sugar over operational semantics (presented in Section 2.1). Recall that operational semantics formalizes how code executes. Especially, it formalizes the possible execution behaviors (terminate, diverge, and stuck). Here, we replace "commands" with "program functions". Such functions can be seen as a partial mapping from inputs to outputs $F : In \to Out$. Given an input $x \in In$, execution of F over x can: (i) terminate and return $y \in Out$, noted $F(x) = y$; (ii) diverge (i.e., never terminate) or raise a runtime error, in that case, F is not defined over x.

Function specifications can be formalized through the Hoare logic [52]. It specifies function behaviors through Hoare triples, noted $\{P\}F\{Q\}$. For total correctness, triple validity intuitively means that for any input x verifying P, running F over x terminates and the result verifies Q. P and Q are then respectively called pre and postcondition.

Definition 4 (Hoare triple)**.** *Let a function F, a predicate P over F inputs and a predicate Q over F outputs. The Hoare triple, noted $\{P\}F\{Q\}$, is valid iff, for all x s.t. $x \models P$, $F(x) = y$ and $y \models Q$.*

A function F can have several preconditions for a given postcondition Q. Still, not all preconditions are useful, some being too restrictive. Thus, we aim for the most generic one, called the *weakest precondition* (WP) [52]. Automatically computing the *weakest precondition* of F w.r.t. Q has been

a strong drive for program analysis since the 70's. Yet, as the whole problem is undecidable, standard approaches must rely on manual annotations or approximations.

Definition 5 (Weakest precondition)**.** *Let a function F and a postcondition Q. The weakest precondition of F w.r.t. Q noted $\mathcal{WP}(F, Q)$ is the most generic precondition, i.e., for all P s.t. $\{P\}F\{Q\}$, $P \Rightarrow \mathcal{WP}(F, Q)$.*

Example 2. *Let* `uint div(uint a, uint b) {return b/a;}` *the function under analysis. Here, $In = Out = [0; 2^n\text{-}1]$ with n the size of an* `uint`*. Note that it is undefined when $a = 0$. Hence, for postcondition $Q_1 = true$, a precondition could be $a = 5$. However, it is too restrictive as other values of a can return safely. The less restrictive one, i.e. $\mathcal{WP}(F, Q_1)$, is $a \neq 0$. Now, consider $Q_2 =$ "the return value must equal 0". Then $\mathcal{WP}(F, Q_2)$ is $a \neq 0 \wedge b < a$.*

Weakest precondition calculus. Automatically computing *weakest preconditions* has been a strong drive for program analysis since the 70's. Given an atomic command, weakest precondition calculus specifies how to compute the weakest precondition through rules given in Table 3.1. Yet, as the whole problem is undecidable, standard approaches must rely on manual annotations, renouncing to full automation. Indeed, the rule handling loops in Table 3.1 needs a loop invariant I to prove specification and a loop variant ν to prove termination. Such annotations must be given by the user, which is a hard task.

Example 3. *Consider the postcondition $Q : x = 10$ for the command c:*

$x := a$;
$i := 0$;
while $i < 10$ *do*
    $x := x + 1$;
    $i := i + 1$;

To handle the loop, WP calculus needs the user to give a loop invariant:

$$I : x = i \wedge i \leq 10$$

and a variant:

$$\nu : 10 - i$$

Then, the WP calculus rule over loops infers that:

$$
\begin{aligned}
& \mathcal{WP}(\textit{while } i < 10 \textit{ do } x := x + 1; i := i + 1, x = 10) \\
\equiv \quad & x = i \wedge i \leq 10 \quad \wedge \quad \forall x', i', \xi, [x' = i' \wedge i' \leq 10 \wedge i' < 10 \wedge \nu = \xi \\
& \Rightarrow \mathcal{WP}(x := x + 1; i := i + 1, x' = i' \wedge i' \leq 10 \wedge \nu < \xi)] \\
& \wedge [x' = i' \wedge i' \leq 10 \wedge i' \geq 10 \Rightarrow x' = 10] \\
\equiv \quad & x = i \wedge i \leq 10 \wedge \forall x', i', \xi, \\
& (x' = i' \wedge i' < 10 \wedge i' - 10 = \xi \Rightarrow x' + 1 = i' + 1 \wedge i' + 1 \leq 10 \wedge 10 - (i' + 1) < \xi) \\
\equiv \quad & x = i \wedge i \leq 10
\end{aligned}
$$

Then we can simply compute:

$$\mathcal{WP}(i := 0, x = i \wedge i \leq 10) \equiv (x = 0)$$

and finally:

$$\mathcal{WP}(x := a, x = 0) \equiv (a = 0)$$

Finally, by using deduction rules of weakest precondition calculus and by giving the loop annotations by hand, we show that $\mathcal{WP}(c, Q) = (a = 0)$.

Table 3.1: Weakest precondition calculus deduction rules

$\mathcal{WP}(skip, Q)$	$\equiv$	Q
$\mathcal{WP}(x := e, Q)$	$\equiv$	$Q[x \leftarrow e]$
$\mathcal{WP}(assert(b), Q)$	$\equiv$	$Q \wedge b$
$\mathcal{WP}(c_1; c_2, Q)$	$\equiv$	$\mathcal{WP}(c_1, \mathcal{WP}(c_2, Q))$
$\mathcal{WP}(\text{if } b \text{ then } c_1 \text{ else } c_2, Q)$	$\equiv$	$(b \Rightarrow \mathcal{WP}(c_1, Q))$ $\wedge \ (\neg b \Rightarrow \mathcal{WP}(c_2, Q))$
$\mathcal{WP}(\text{while } b \text{ invariant } I \text{ variant } \nu, \prec \text{ do } c, Q)$	$\equiv$	$I \wedge \forall x_1, ..., x_k, \xi,$ $(I \wedge b \wedge \xi = \nu \Rightarrow \mathcal{WP}(e, I \wedge \xi \prec \nu))$ $\wedge \ (I \wedge \neg b \Rightarrow Q)$

Where x_j are references modified in the loop body.

3.2.2 Constraint Acquisition

The constraint acquisition (CA) process can be seen as an interplay between the learner and the user. For that, the learner needs to share some common *vocabulary* to communicate with the user. This vocabulary is a finite set of

variables X taking values in a finite domain D. A constraint c is defined on a subset of variables and a relation specifying which values are allowed. A *constraint network* is a set C of constraints. An example $e \in D^{|X|}$ *satisfies* a constraint c if the projection of e on c variables is in c. An example e *is a solution of* C if and only if it satisfies all constraint in C.

In addition to the vocabulary, the learner owns a *language* Γ of bounded arity relations from which it can build constraints on specified sets of variables. The *constraint bias*, denoted by B, is a set of constraints built from Γ on (X, D), from which the learner builds a constraint network. A *concept* is a Boolean function f over D^X. A *representation* of a concept f is a constraint network C for which $f^{-1}(true)$ equals the solutions set of C. A *membership query* takes an example e and asks the user to classify it. The answer is *yes* iff, e is a solution of the user concept. For any example e, $\kappa(e)$ denotes the set of all constraints in B rejecting e.

We now define *convergence*. Given a set E of examples labelled by the user *yes* or *no*, we say that a network C agrees with E if C accepts all examples labelled *yes* in E and does not accept those labelled *no*. The learning process has *converged* on the network $L \subseteq B$ if **(i)** L agrees with E and **(ii)** for every other $L' \subseteq B$ agreeing with E, we have $L' \equiv L$.

Conacq is a CA system that submits *membership queries* to a user. Conacq uses a concise representation of the learner's version space into a clausal formula. Formally, any constraint $c \in B$ is associated with a Boolean atom $a(c)$ stating if c must be in the learned network. Conacq starts with an empty theory and iteratively expands it by generating and submitting to the user an informative example. An informative example ensures to reduce the learner's version space independently from the user answer. If no informative example remains, this means that we converged and Conacq returns the theory encoding the learned network.

Example 4. *Consider the set of variables* $X = \{x, y\}$ *defined over* $\{-10..10\}$. *We aim to infer a constraint network equivalent to* $C_T = x > 0 \wedge y > 0$. *Consider the bias* $B = \{x > 0, x \leq 0, y > 0, y \leq 0, x = y, x \neq y\}$. *The execution of* Conacq *is described in Table 3.2. Observe that only informative queries are generated – e.g., after* e_1 *no query where* $x > 0$, $y > 0$ *and* $x = y$ *will be generated as it would be redundant.*

After four queries, there is no other informative query left and the version space T *is equal to:* $\neg a(x \leq 0) \wedge \neg a(y \leq 0) \wedge \neg a(x \neq y) \wedge \neg a(x = y) \wedge [a(x \leq 0) \vee a(y > 0) \vee a(x = y)] \wedge [a(x > 0) \vee a(y \leq 0) \vee a(x = y)]$. *By unit propagation,* T *reduces to* $\neg a(x \leq 0) \wedge \neg a(y \leq 0) \wedge \neg a(x \neq y) \wedge \neg a(x = y) \wedge a(y > 0) \wedge a(x > 0)$.

Thus, Conacq *converged to the network* $L = x > 0 \wedge y > 0$ *which is indeed equivalent to* C_T.

CA is PBE. Usually, constraint acquisition is not presented as programming by example – presented in Section 2.4. Indeed, it comes from the constraint

Queries	Answer	Version space (T)
$e_1 = (x \leftarrow 1, y \leftarrow 1)$	yes	$T_1 = \neg a(x \leq 0) \wedge \neg a(y \leq 0) \wedge \neg a(x \neq y)$
$e_2 = (x \leftarrow 2, y \leftarrow -1)$	no	$T_2 = T_1 \wedge (a(x \leq 0) \vee a(y > 0) \vee a(x = y))$
$e_3 = (x \leftarrow 1, y \leftarrow 3)$	yes	$T_3 = T_2 \wedge \neg a(x = y)$
$e_4 = (x \leftarrow -1, y \leftarrow 3)$	no	$T_4 = T_3 \wedge (a(x > 0) \vee a(y \leq 0) \vee a(x = y))$

Table 3.2: Example of CONACQ execution

programming community and not from the program synthesis one. Still, it integrates well in the programming by example (PBE) framework. Indeed, it can be seen as a special case of active PBE where the search space contains only conjunctions of constraints taken in a finite set – no more context-free grammar, nor infinite space hypothesis. Such limitations enable, as we will see in Section 3.4.3, to enjoy clear correctness guarantees.

3.3 Motivation

We focus on *memory-related preconditions* – e.g., predicates stating on which inputs a function can be executed without leading to a memory violation – in a *black-box manner*. Let us consider the prototype of function `find_first_of` in Listing 3.1 (from Frama-C [108] test suite). We aim to infer which values of `a`, `m`, `b` and `n` are accepted without relying on the source code – still we can execute the code over chosen input and observe results.

```
1 void find_first_of(int* a, int m, int *b, int n)
```

Listing 3.1: Function prototype

From white-box to black-box. White-box analysis (such as P-Gen [100]) uses the program source code to infer preconditions. Yet several practical scenarios are impractical for white-box methods. First, having the *whole* source code is often unrealistic (many projects embed third-party components). Second, in practice program analyzers focus on a single programming language, but many projects use combinations of them (e.g., inline assembly in C code). Third, despite huge progress in the past decades, white-box program analysis still suffers on large or complex codes (unbounded loops, recursion, dynamic allocation, etc.) possibly leading to serious scalability or precision issues. Fourth, obfuscation is common in certain ecosystems to make reverse engineering harder and thwart white-box analysis. In all these scenarios, black-box methods are the sole option (cf. experiments on Section 3.6, RQ4). Yet, as generalization is involved, black-box methods can compute *incorrect* preconditions (i.e., formula actually being not preconditions).

Black-box passive learning is not enough. Black-box methods should exercise the function under analysis on a representative set of test cases to infer relevant preconditions. A solution is to assume that users can provide such

tests and leverage passive learning. Yet, this is often unrealistic – especially when the source code is not available. Moreover, random testing is rarely satisfactory, e.g., with 100 random test cases, both Daikon [40] and PIE [70] infer here an *incorrect* precondition for `find_first_of`.

Active learning. Gehr et al. [68] performs active learning, generating test cases automatically. Such approaches are more actionable and less sensitive to user bias. Still, methods developed so far lack theoretical guarantees. Indeed, they cannot ensure that all useful test cases have been considered. Gehr et al. method infers in $\approx 700s$ an *incorrect* precondition for `find_first_of`, generating 177 test cases.

PreCA insights. Our method performs black-box *precondition inference* through active constraint acquisition [122]. Unlike previous active approaches, PreCA mixes the sampling and learning phases which enables to show good theoretical properties. Indeed, when a test case is generated, PreCA directly observes how the function behaves on it and updates its search space accordingly. As such, given a set of constraints B called the bias, PreCA will generate all test cases to ensure convergence modulo B. Thus, if all queries can be exactly classified and if B is expressive enough, PreCA returns the *weakest precondition*. Regarding our example, it infers the (correct) *weakest precondition* $(m > 0 \Rightarrow valid(a)) \wedge (m > 0 \wedge n > 0 \Rightarrow valid(b))$ where $valid(p) \equiv (p \neq NULL)$ in 172s, running 45 test cases. The implementation of `find_first_of` is given in Listing 3.2, for readers to check the precondition.

```
1  int find(int* t, int n, int val)
2  {
3      for (int i = 0; i < n; i++)
4      {
5          if (t[i] == val)
6              return i;
7      }
8      return n;
9  }
10
11 int find_first_of(int* a, int m, int* b, int n)
12 {
13     for (int i = 0; i < m; i++)
14     {
15         if (find(b, n, a[i]) < n)
16             return i;
17     }
18     return m;
19 }
```

Listing 3.2: Implementation of find_first_of

Table 3.3: `find_first_of` results, no source code

Alg.	Active?	Success.	#Test cases	Time
Daikon	no	no	100	0.6s
PIE	no	no	100	11s
Gehr et al.	yes	no	177	700s
P-Gen	(white-box)	(do not apply)	-	-
PreCA	**yes**	**yes**	**45**	**172s**

3.4 Precondition Acquisition

Given a function under analysis F, we aim to infer the *weakest precondition* of F w.r.t. some postcondition Q through CA. Note that, as generalization is involved, we are not sure *a priori* to compute a real precondition, hence the wording "likely-precondition" introduced in Daikon [40]. Guarantees are studied in Section 3.4.3. To our knowledge, this is the first time CA is used for program analysis.

3.4.1 Problem at Hand

We show here that precondition inference can be translated to a CA problem. In our context the user is replaced by an *oracle*, which automatically answers queries – implemented in practice by running the program on given inputs – and the *target concept* is $\mathcal{WP}(F, Q)$. The set of *variables* X equals (M) where M is the initial memory state to run F. M is a map from symbols – like F arguments and global variables – to their values and the *domain* D of M is the finite set of all its possible mappings – thus, D^X equals In, the definition domain of F. The *constraint language* Γ and the *bias* B are sets of constraints over M. We describe precisely Γ and B in Section 3.5.1. Finally, a *membership query* e is a complete assignment of M s.t. F can be executed over e.

3.4.2 Description of PreCA

We detail here our approach, dubbed PreCA, composed of 1. the *oracle*; 2. the *acquisition module*.

Oracle. Given a function F and a postcondition Q, PreCA queries an oracle to classify membership queries. It takes F, Q and an input $e \in In = D^X$ and answers in finite time. The oracle must comply to the following specification:

$$\texttt{runOracle}(F, Q, e) = \begin{cases} \textit{yes} \text{ or } \textit{ukn} \text{ if } e \models \mathcal{WP}(F, Q) \\ \textit{no} \text{ or } \textit{ukn} \text{ otherwise} \end{cases}$$

Note that the oracle answers *ukn* when it cannot classify e, extending the Conacq framework where the user must answer only by *yes* or *no*. In practice,

Algorithm 1: PreCA

In : A function F; a postcondition Q; a bias B; variables X;
Out : A constraint network over F input encoded by Ω consistent with oracle answers or collapse;

```
1  begin
2      Ω ← ⊤
3      while true do
4          e ← QueryGeneration(B, X, Ω)
5          if e = nil then
6              if Ω is SAT then
7                  return network(Ω)
8              else return "collapse"
9          if runOracle(F, Q, e) ≠ yes then
10             Ω ← Ω ∧ (⋁_{c∈κ(e)} a(c))
11         else Ω ← Ω ∧ (⋀_{c∈κ(e)} ¬a(c))
```

such oracle runs F over e with a timeout. If the execution timeouts it returns (i) *ukn*, otherwise it returns (ii) *yes* if $F(e) = y$ and $y \models Q$; (iii) *no* if $F(e) = y$ and $y \not\models Q$ or if execution raises a runtime error.

Acquisition module. PreCA (see Algo.1) starts from an empty theory Ω and iteratively expands it by processing examples generated at line 4. PreCA submits these examples to the oracle for a classification (`runOracle` call at line 9). If the oracle answers *yes*, we must discard all constraints of B in $\kappa(e)$, those rejecting e, by expanding Ω with negative unit clauses (line 11). However, if the oracle answers *no* or *ukn*, Ω is expanded with a clause consisting of all literals $a(c)$ s.t. $c \in \kappa(e)$ (line 10). Bear in mind that `QueryGeneration` function returns informative examples aiming to reduce Ω to a monomial (conjunction of unit clauses). `QueryGeneration` is used exactly as it appears in [122]. If there is no example to return, this means that Ω is monomial. Now if Ω is not satisfiable, a *"collapse"* message is returned (line 8). This happen when the concept to learn is not representable by B. Otherwise, we return the constraint network encoded by Ω through the `network` function (line 7).

3.4.3 Theoretical Analysis

We show that PreCA terminates and that learned preconditions are sound when PreCA is fed with an expressive enough bias B. Then we show that if `runOracle` never answers *ukn*, PreCA returns the weakest precondition.

Proposition 1 (Consistency)**.** *Given a function F, a postcondition Q and a bias B. If* PreCA *returns a network L, then L agrees with all positive and negative queries.*

Proof. Let e a positive query. When processing e, PreCA discards all constraints $c \in \kappa(e)$. Thus, if PreCA returns a network L, L contains no constraint from $\kappa(e)$. Thus, $e \models L$. Consider now that e is negative. When processing e, PreCA will extend Ω with the clause $\bigvee_{c \in \kappa(e)} a(c)$. If PreCA returns L, this mean that at least one constraint from $\kappa(e)$ has been integrated to L. Thus, $e \not\models L$. □

Proposition 2 (Termination)**.** *Given a function F, a postcondition Q and a bias B,* PreCA *terminates.*

Proof. Termination of PreCA immediately follows the reduction of Ω to a monomial with an atom for each constraint $c \in B$. Ω is expanded with negative unit clauses of the form $\neg a(c)$ or by non-unit clauses composed with positive atoms $a(c)$. An informative example is an example expanding Ω with new atoms or reducing existing non-unit clauses size. As (i) Ω involves a finite number of atoms (B being a finite set of constraints), (ii) `QueryGeneration` terminates returning an informative example if it exists, *nil* otherwise, and (iii) `runOracle` always responds, we have termination. □

Proposition 3 (Soundness)**.** *Given a function F, a postcondition Q and a bias B s.t. $\mathcal{WP}(F, Q)$ is representable by B. If* PreCA *returns a network L then L is a precondition of F w.r.t. the postcondition Q.*

Proof. L is a precondition means that $L \Rightarrow \mathcal{WP}(F, Q)$. Suppose that $L \not\Rightarrow \mathcal{WP}(F, Q)$ when PreCA do not collapse. This means that there exists $e \models L$ and $e \not\models \mathcal{WP}(F, Q)$. As $\mathcal{WP}(F, Q)$ is representable by B, it exists $c^* \in B$ that rejects e. As PreCA terminates, this means that `QueryGeneration` is not able to generate an example because Ω is reduced to a monomial. Knowing that $e \models L$ and $e \not\models c^*$, it exists a unit clause of the form $\neg a(c^*)$ in Ω learned by processing a positive example e^* s.t. $e^* \not\models c^*$. However, as $c^* \in \mathcal{WP}(F, Q)$, such e^* does not exist. It proves that $L \Rightarrow \mathcal{WP}(F, Q)$. □

Theorem 1 (Correctness)**.** *Given a function F, a postcondition Q and a bias B s.t. $\mathcal{WP}(F, Q)$ is representable by B. If* `runOracle` *never returns ukn then* PreCA *converges to a network L equivalent to the weakest precondition.*

Proof. If $\mathcal{WP}(F, Q) \subseteq B$ and `runOracle` returns *yes*/*no* answers, PreCA is equivalent to Conacq. Conacq is correct, terminates and always converges when B is expressive enough [122], it follows that PreCA always converges on to a constraint network L equivalent to $\mathcal{WP}(F, Q)$ under the assumptions on B and `runOracle`. □

Discussion. These guarantees, while not perfect, are still very pleasant for a black-box approach. Prior black-box learners are much more limited: Daikon [40] does not guarantee consistency (Proposition 1), while [70, 68] guarantee consistency but not correctness (Theorem 1). Also, previous black-box methods consider that functions always terminate i.e., no *ukn* answers.

3.5 PreCA for Memory-oriented Preconditions

We now setup PreCA to the case of *memory-related preconditions*, which are of paramount importance for the safety and security of low-level languages like C or binary code.

3.5.1 Constraint Acquisition Settings

Vocabulary *(X, D).* Given a function F, our variables set $X = \{p_1, \ldots, p_k, i_1, \ldots, i_{k'}\}$ represents the initial memory state of F. It is composed of all F arguments and global variables in scope. Here, p_j are pointers and i_j are integers (signed or not). D^X defines possible F inputs. It compactly represents all cases induced by Γ. We note $r_1, ..., r_m$ the address of each global variables in X and $a_1, ..., a_k$, k pairwise distinct new valid addresses. Then, $D(p_i)$ is $\{NULL, r_1, ..., r_m, a_1, ..., a_j\}$ and $D(i_j)$ is $[\,0\,,\, N_U\,]$ if i_j is unsigned and $[\,-N_I\,,\, N_I\,]$ otherwise – N_I and N_U are the number of signed and unsigned integers in X.

Language Γ. PreCA considers the constraint language Γ described in Table 3.4 including well-typed constraints only. Observe that: (i) it does not include conjunctions of constraints as acquisition will infer them; (ii) Γ holds Horn clauses of arbitrary size which is crucial to handle conditional preconditions, e.g., `find_first_of` *weakest precondition* in Listing 3.1 contains the constraint $m > 0 \Rightarrow valid(a)$.

Bias *B*. The bias B is a finite set of constraints extracted from Γ. A balance must be found here, as a large bias is more expressive but can slow down inference. Given the function F, PreCA considers the following heuristic: *"Let i be the number of F integer inputs and $k = max(i, 1)$. Then* PreCA *bias includes all Horn clauses of size $\leq k + 1$ from Γ"*. Indeed, from our experience, validity of a pointer is usually conditioned by constraints over integer variables.

3.5.2 Speeding up PreCA

First, we describe PreCA background knowledge. Secondly, we present a domain-based preprocessing heuristic.

Table 3.4: Grammar of constraint language Γ

Grammar		
P	:=	$C \Rightarrow A \mid A \mid \neg A$
C	:=	$C \wedge C \mid A \mid \neg A$
A	:=	$valid(p_j) \mid alias(p_j, p_l) \mid deref(p_j, g)$
	$\mid$	$i_j = 0 \mid i_j < 0 \mid i_j \leq 0 \mid i_j = i_l \mid i_j < i_l \mid i_j \leq i_l$
Semantics of constraint over pointers		
$valid(p_j)$	$\equiv$	$p_j \neq NULL$
$alias(p_j, p_l)$	$\equiv$	$p_j = p_l$
$deref(p_j, g)$	$\equiv$	$p_j = \&g$ where $\&g$ is the address of g

p_j (resp. i_j) are pointers (resp. integers) and g is a global variable.

Background knowledge. A background knowledge K to speed up convergence of CA contains known relations over the bias constraints to filter incoherent networks. Table 3.5 shows a subset of K. It contains usual boolean properties, transitivity relations over integers and relations on memory – e.g., if p_1 is valid and p_1 aliases with p_2 then p_2 is valid.

$$
\begin{array}{|l}
a(c) \longrightarrow \neg a(\bar{c}),\ \forall c \in B \\
a(c_1) \longrightarrow a(c_1 \vee c_2)\ ,\ \forall c_1, c_2 \in B \\
a(i_1 = 0) \wedge a(i_1 = i_2) \longrightarrow a(i_2 = 0) \\
a(i_1 = i_2) \wedge a(i_2 = i_3) \longrightarrow a(i_1 = i_3) \\
a(\neg valid(p_1)) \wedge a(\neg alias(p_1, p_2)) \longrightarrow a(valid(p_2)) \\
a(valid(p_1)) \wedge a(alias(p_1, p_2)) \longrightarrow a(valid(p_2)) \\
a(alias(p_1, p_2)) \wedge a(alias(p_2, p_3)) \longrightarrow a(alias(p_1, p_3))
\end{array}
$$

Where p_j (resp. i_j) are pointer (resp. integer) variables.

Table 3.5: Background knowledge K (a subset)

Preprocess. Functions rarely raise runtime errors or contradict postconditions over valid and non aliasing pointers (i.e., the easy case that programmers usually handle well). Thus, *given a function F, we call likely-positive queries assignments of F inputs s.t. at most one p_j is invalid or at most one pair (p_j, p_l) aliases.* Over *likely-positive queries*, the oracle will probably answer *yes* which would be really helpful as it would discard all constraints from $\kappa(e)$ (unlike negative ones which introduce non-unit clause in Ω, see Algorithm 1). Thus, PRECA starts by *likely-positive queries* in the hope to prune the search space before launching the active phase.

3.6 Experimental Evaluation

We implemented PreCA[1] in JAVA, and rely on the Choco constraint solver and MiniSat SAT solver. We evaluate PreCA on the following Research Questions:

RQ1 *Can* PreCA *handle realistic functions?* We launch PreCA against our benchmark and check if it indeed infers *weakest preconditions*;

RQ2 *How* PreCA *components influence results?* We compare PreCA with and without background knowledge, preprocess and active learning;

RQ3 *Is* PreCA *competitive with black-box methods?* We compare to black-box state-of-the-art methods in terms of correctness and speed;

RQ4 *Is* PreCA *competitive with white-box methods?* We compare to the white-box method P-Gen on clean C code, and consider also 3 "hard" scenarios: no source code, obfuscated code, presence of inline assembly.

3.6.1 Experimental Design

Benchmark. Our benchmark considers 50 real C functions. It contains all functions from string.h, all functions from [100, 69] (except 2 functions from an old Xen version), functions from the DSA benchmark (`https://tinyurl.com/tvzzpvmm`), Frama-C WP test suite (`https://tinyurl.com/ycxdbjf3`), Siemens suite [124] and the book Science of Programming [125]. Functions range from 3 LoC to 250 (mean 59), have up to 9 loops (mean 2.8, 47/50 functions with loops) and 2/50 with recursive calls.

Postconditions. For each function, we study two scenarios: with the implicit true postcondition (dubbed "no postcondition") and with explicit postcondition. In the latter case, we manually choose relevant ones, e.g. $Q = valid(ret)$ for pointers, and $Q = ret \neq 0$ or $Q = ret > 0$ for integers. Finally, six functions returns no output and are discarded. In total, our benchmark contains 94 inference tasks, 50 with the implicit postconditions and 44 with explicit postconditions.

Setup. We run PreCA with different time budgets per function (from 1s to 1h) and an oracle timeout of 1min (leading to the *ukn* answers). Experiments are done on a machine with 6 Intel Xeon E-2176M CPUs and 32 GB of RAM.

3.6.2 Experimental Results

Results are summarized in Table 3.6.

RQ1. With a time budget of 5min per example and without postcondition, PreCA infers 46/50 *weakest preconditions* (29/50 for 1s, 38/50 for 5s). Two examples timeout, and two others return a constraint network not equivalent

[1] `https://github.com/binsec/preca`

to the *weakest precondition* – a manual inspection shows our bias is not expressive enough in these cases, still it returns a (correct) precondition for one of them.

With postconditions, PreCA infers 18/44 *weakest preconditions* with < 5 min time budget each (11/44 for 1s, 16/44 for 5s) and never timeouts (in 7 additional cases it still returns a correct precondition). All these results are far better than other state-of-the-art tools (**RQ3**, **RQ4**).

PreCA *is able to handle real functions precisely (weakest precondition) in a small amount of time. Especially, it is extremely accurate for implicit postconditions.*

RQ2. First, we consider PreCA in passive mode, with 100 random queries, in order to see the impact of active learning (denoted ↳ Random in Table 3.6). Results are averaged over 10 runs per function. We see a significant drop in performance for time budgets $\geq$ 5min (for 5min: 30/50 vs 46/50, 18/44 vs 12/44). Second, we study how the background knowledge and the preprocess impact PreCA results. We see a clear impact only for small time budgets (e.g., 1s and no postcondition: 29 vs 15/19/13). Interestingly, both the background knowledge and the preprocess are necessary to get speedup.

PreCA *benefits strongly from its active mode. Background knowledge and preprocess over complex preconditions are useful for small time budgets.*

RQ3. We compare now against state-of-the art black-box precondition learners, namely Daikon [40], PIE [70] and Gehr et al. approach [68] – code being unavailable, we reimplemented it. Daikon and PIE performing passive learning, we run them over 100 random queries. As Daikon, PIE and Gehr et al. methods are randomized, we run them 10× and report their average results. We first observe that PreCA performs significantly better than these three competitors for all setups – for 1s and no postcondition: 29 vs 8.0 - 16.0 - 1.4; for 1h and no postcondition: 46 vs 26.1 - 17.7 - 1.6. We tried feeding Daikon, PIE and Gehr et al. with PreCA queries (lines ↳ PreCA and ↳ Both). All methods except Daikon benefit from it, highlighting the quality of PreCA sample generation mechanism.

PreCA *significantly outperforms prior black-box methods. Especially, it infers in 5s more weakest preconditions than Daikon, PIE and Gehr et al. in 1h. Moreover, it generates high quality queries that can benefit other methods.*

RQ4. We compared to the white-box method P-Gen [100]. We also considered [126] and [127], but the former requires to manually translate C code to Prolog (no front-end provided) and the latter is not available. First, we consider a favourable setup where the source code of our 94 examples is available (Table 3.6). Surprisingly, PreCA infers slightly more WP with a 5s time budget than P-Gen with 1h (both with and without postcondition). The gap increases for a time budget of 1h and implicit postconditions (46 vs 37). Second, we consider "hard" application scenarios: (i) no source code; (ii)

obfuscated code; (iii) inline assembly – our 94 samples are transformed accordingly. As expected for a white-box method, P-Gen infers no precondition for these scenarios (0/94) while PreCA results remain the same.

As expected, PreCA *significantly outperforms P-Gen on hard application scenarios. Less expected, it performs also better in the case where the source code is fully available.*

Table 3.6: Results with and without postconditions depending on the time budget

	1s		5s		5 mins		1h	
	$\#WP_\top$	$\#WP_Q$	$\#WP_\top$	$\#WP_Q$	$\#WP_\top$	$\#WP_Q$	$\#WP_\top$	$\#WP_Q$
Daikon	1.4/50	0.4/44	1.6/50	0.4/44	1.6/50	0.4/44	1.6/50	0.4/44
↳ PreCA	2/50	1/44	2/50	1/44	2/50	1/44	2/50	1/44
↳ Both	3.3/50	0/44	5.7/50	0/44	5.7/50	0/44	5.7/50	0/44
PIE	16.4/50	4.7/44	16.4/50	4.7/44	17.7/50	4.7/44	17.7/50	5.3/44
↳ PreCA	5/50	3/44	5/50	3/44	5/50	3/44	5/50	3/44
↳ Both	25.3/50	11.3/44	25.4/50	11.3/44	26.4/50	11.3/44	28.4/50	11.3/44
Gehr et al.	8.0/50	5.0/44	16.8/50	8.1/44	26.1/50	10.1/44	26.1/50	10.3/44
↳ PreCA	37/50	15/44	43/50	17/44	46/50	18/44	46/50	18/44
PreCA	29/50	11/44	**38/50**	**16/44**	**46/50**	**18/44**	**46/50**	**18/44**
↳ BK	15/50	8/44	38/50	16/44	45/50	18/44	46/50	18/44
↳ Preproc.	19/50	9/44	36/50	16/44	45/50	18/44	46/50	18/44
↳ $\emptyset$	13/50	7/44	35/50	15/44	45/50	18/44	46/50	18/44
↳ Random	29.9/50	12.1/44	29.9/50	12.1/44	30.0/50	12.1/44	30.0/50	12.1/44
P-Gen	**34/50**	**13/44**	37/50	15/44	37/50	15/44	37/50	15/44

$\#WP_\top$ (resp. $\#WP_Q$) is the number of inferred weakest precondition without (resp. with) a postcondition. We study 3 variations of Daikon and PIE: (i) original one (highlighted) on 100 random examples; (ii) on PreCA examples; (iii) on both random and PreCA examples. We study the original active Gehr et al. method (highlighted) and we feed it with PreCA examples. Finally, we study PreCA with its background knowledge and preprocess (highlited), with background knowledge only (BK), with preprocessing only (Preproc.), without any of them ($\emptyset$) and in passive mode with 100 random queries (Random). P-Gen being a static method, we consider only its original form.

3.6.3 Discussion

While PreCA shows overall good properties, it also comes with a few limitations. First, handling constant values is problematic. Indeed, we should add comparisons to them in the bias. However, in a black-box context, there is no reason to choose one constant value from another and we cannot add all of them as bias would explode. Second, PreCA uses Horn clauses to handle disjunctive specifications. We consider a simple heuristic for size selection (Section 3.5.1), yet a more principled approach is desirable. Finally, we require the function under analysis to be deterministic (a common assumption in the field). Going further remains open.

3.7 Related Work

Black-box contracts inference. Daikon [40] dynamically infers preconditions through predefined patterns over the evolution of variable values. The technique is passive and lacks clear foundations. PIE [70] relies on program synthesis for black-box precondition inference. Garg et al. [128] and Sankaranarayanan et al. [69] infer invariants and preconditions through tree learning algorithms. As invariant inference distinguishes from precondition inference, we did not consider [128] in our evaluation. However, even if [69] method was not available, we integrated their use-cases and show that we handle them all (except one) while enjoying better theoretical properties. These methods perform passive learning and heavily depend on test cases quality. Gehr et al.'s method [68] relies on black-box active learning. Yet, it relies on program synthesis and performs (*type-aware*) random sampling, preventing it to enjoy PreCA correctness properties.

White-box dynamic contracts inference. While purely static white-box approaches [119, 129, 127, 126] are considered imprecise (too conservative) and hard to get right (loops, memory, etc.), some approaches combine dynamic reasoning together with white-box information. Seghir et al. [100] method must translate the analyzed function into transition constraints being thus highly impacted by code complexity (Section 3.6.2 RQ4). On the other hand, Astorga et al. [120, 96] relies on symbolic execution to retrieve a set of useful inputs and language features, yet the technique is incomplete in the presence of loops and cannot ensure that all interesting test cases were tested.

Constraint acquisition. CA has been applied to different contexts from scheduling [36] to robotics [37]. However, this is the first time CA is applied to program analysis and precondition inference. While we rely on Conacq, other techniques exist [36, 130, 131] and could be explored.

Program synthesis. Program synthesis [132] aims at creating a function meeting a given specification, given either formally, in natural language or *as input-output relations*. This last case shows some similarities with precondition inference and is used in some prior work on black-box inference [68, 70].

3.8 Conclusion

We propose the first application of Constraint Acquisition to the Precondition Inference problem, a major issue in Program Analysis and Formal Methods. We show how to instantiate the standard framework to the program analysis case, yielding the first black-box active precondition inference method with clear guarantees. Moreover, our experiments for memory-oriented preconditions show that PreCA significantly outperforms prior works, demonstrating

the interest of Constraint Acquisition here. Thus, our new method fulfil all the properties presented in Section 1.3: it is completely black-box, can work at binary-level, is fast and enjoys clear correctness properties.

Chapter 4

Synthesizing code semantics

Abstract

Code obfuscation aims at protecting Intellectual Property and other secrets embedded into software from being retrieved. Recent works leverage advances in artificial intelligence (AI) with the hope of getting black-box deobfuscators completely immune to standard (white-box) protection mechanisms. While promising, this new field of *AI-based*, and more specifically *search-based black-box deobfuscation*, is still in its infancy. In this article we deepen the state of search-based black-box deobfuscation in three key directions: *understand* the current state-of-the-art, *improve* over it and design dedicated *protection mechanisms*. In particular, we define a novel generic framework for search-based black-box deobfuscation encompassing prior work and highlighting key components; we are the first to point out that the search space underlying code deobfuscation is too unstable for simulation-based methods (e.g., Monte Carlo Tree Search used in prior work) and advocate the use of robust methods such as S-metaheuristics; we propose the new optimized search-based black-box deobfuscator Xyntia which significantly outperforms prior work in terms of success rate (especially with small time budget) while being completely immune to the most recent anti-analysis code obfuscation methods; and finally we propose two novel protections against search-based black-box deobfuscation, allowing to counter Xyntia powerful attacks.

4.1 Introduction

Software contain valuable assets, such as secret algorithms, business logic or cryptographic keys, that attackers may try to retrieve. The so-called Man-At-The-End-Attacks scenario (MATE) considers the case where software users themselves are adversarial and try to extract such information from the code. *Code obfuscation* [77, 11] aims at protecting codes against such attacks, by transforming a sensitive program P into a functionally equivalent program P' that is more "difficult" (more expensive, for example, in money or time) to

understand or modify. On the flip side, *code deobfuscation* aims to extract information from obfuscated codes.

White-box deobfuscation techniques, based on advanced symbolic program analysis, have proven extremely powerful against standard obfuscation schemes [133, 134, 135, 136, 30, 137, 31] – especially in local attack scenarios where the attacker analyzes pre-identified parts of the code (e.g., trigger conditions). But they are inherently sensitive to the *syntactic complexity* of the code under analysis, leading to recent and effective countermeasures [32, 33, 138, 77].

Search-based black-box deobfuscation. Despite being rarely complete or sound, *artificial intelligence* (AI) techniques are flexible and often provide good enough solutions to hard problems in reasonable time. They have been therefore recently applied to binary-level code deobfuscation. The pioneering work by Blazytko et al. [67] shows how *Monte Carlo Tree Search* (MCTS) [139] can be leveraged to solve local deobfuscation tasks by *learning* the semantics of pieces of protected codes in a *black-box manner*, in principle *immune to the syntactic complexity* of these codes. Their method and prototype, Syntia, have been successfully used to reverse state-of-the-art protectors like VMProtect [140], Themida [141] and Tigress [142], drawing attention from the software security community [143].

Problem. While promising, search-based black-box (code) deobfuscation techniques are still not well understood. Several key questions of practical relevance (e.g., deobfuscation correctness and quality, sensitivity to time budget) are not addressed in Blazytko et al.'s original paper, making it hard to exactly assess the strengths and weaknesses of the approach. Moreover, as Syntia comes with many hard-coded design and implementation choices, it is legitimate to ask whether other choices lead to better performance, and to get a broader view of search-based black-box deobfuscation methods. Finally, it is unclear how these methods compare with recent proposals for greybox deobfuscation [73] or general program synthesis [91, 10], and how to protect from such black-box attacks.

Goal. We focus on advancing the current state of search-based black-box deobfuscation in the following three key directions: (1) generalize the initial Syntia proposal and refine the initial experiments by Blazytko et al. in order to better *understand* search-based black-box methods, (2) *improve* the current state-of-the-art (Syntia) through a careful formalization and exploration of the design space and evaluate the approach against greybox and program synthesis methods, and finally (3) study how to *mitigate* such black-box attacks. Especially, we study the underlying search space, bringing new insights for efficient black-box deobfuscation, and promote the application of S-metaheuristics [144] instead of MCTS.

Contributions. Our main contributions are the following:

- We refine Blazytko et al.'s experiments in a *systematic way*, highlighting *new strengths and new weaknesses* of the initial SYNTIA proposal for search-based black-box deobfuscation (Section 4.4). Especially, SYNTIA (based on Monte Carlo Tree Search) is far less efficient than expected for small time budgets (typical usage scenario) and lacks robustness;

- We propose a missing *formalization of black-box deobfuscation* (Section 4.4) and dig into SYNTIA internals to rationalize our observations (Section 4.4.4). It appears that *the search space underlying black-box code deobfuscation is too unstable* to rely on MCTS – especially assigning a score to a *partial state* through *simulation* leads to poor estimations. As a result, SYNTIA is here *almost enumerative*;

- We propose to see black-box deobfuscation as an *optimization problem* rather than a *single player game* (Section 4.5), allowing to reuse *S-metaheuristics* [144], known to be more robust than MCTS on unstable search spaces (especially, they do not need to score partial states). We propose XYNTIA (Section 4.5), a *search-based black-box deobfuscator* using *Iterated Local Search* (ILS) [145], known among S-metaheuristics for its robustness. Experiments show that XYNTIA keeps the benefits of SYNTIA while correcting most of its flaws. Especially, XYNTIA *significantly outperforms* SYNTIA, synthesizing twice more expressions with a budget of 1 s/expr than SYNTIA with 600 s/expr. Other S-metaheuristics also clearly beat MCTS, even if they are less effective here than ILS;

- We evaluate XYNTIA against other *state-of-the-art attackers* (Section 4.6), namely the QSYNTH greybox deobfuscator [73], program synthesizers CVC4 [91] and STOKE [10], and pattern-matching-based simplifiers. XYNTIA outperforms all of them – it finds 2× more expressions and is 30× faster than QSYNTH on heavy protections;

- We evaluate XYNTIA against *state-of-the-art defenses* (Section 4.7), especially recent anti-analysis proposals [138, 12, 32, 146, 83]. As expected, XYNTIA is immune to them. In particular, it successfully bypasses side-channels [83], path explosion [32] and MBA [138]. It also synthesizes VM-handlers from state-of-the-art virtualizers [142, 140];

- Finally, we propose the *two first protections against search-based black-box deobfuscation* (Section 4.8). We observe that all phases of black-box techniques can be thwarted (hypothesis, sampling and learning), we propose two practical methods exploiting these limitations and we discuss them in the context of virtualization-based obfuscation: (i) *semantically complex handlers*; (ii) *merged handlers with branch-less conditions.* Experiments show that both protections are highly effective.

We hope that our results will help better understand search-based deobfuscation, and lead to further progress in this promising field.

4.2 Background

4.2.1 Obfuscation

Program obfuscation [77, 11] is a family of methods designed to make reverse engineering (understanding programs internals) hard. It is employed by manufacturers to protect intellectual property and by malware authors to hinder analysis. It transforms a program P in a functionally equivalent, more complex program P' with an acceptable performance penalty. Obfuscation does not ensure that a program cannot be understood – this is impossible in the MATE context [81] – but aims to delay the analysis as much as possible in order to make it unprofitable. Thus, it is especially important to protect from *automated deobfuscation analyses* (anti-analysis obfuscation). We present here two important obfuscation methods.

Mixed Boolean-Arithmetic (MBA) encoding [138] transforms an arithmetic and/or Boolean expression into an equivalent one, combining arithmetic and Boolean operations. It can be applied iteratively to increase the syntactic complexity of the expression. Eyrolles et al. [147] show that SMT solvers struggle to answer equivalence requests on MBA expressions, preventing the automated simplification of protected expressions by symbolic methods.

Virtualization [146] translates an initial code P into a bytecode B together with a custom virtual machine. Execution of the obfuscated code can be divided in 3 steps (Fig. 4.1): (i) *fetch* the next bytecode instruction to execute, (ii) *decode* the bytecode and find the corresponding *handler*, (iii) and finally *execute* the handler. Virtualization hides the real control-flow-graph (CFG) of P, and reversing the handlers is key for reversing the VM. Virtualization is notably used in malware [148, 149].

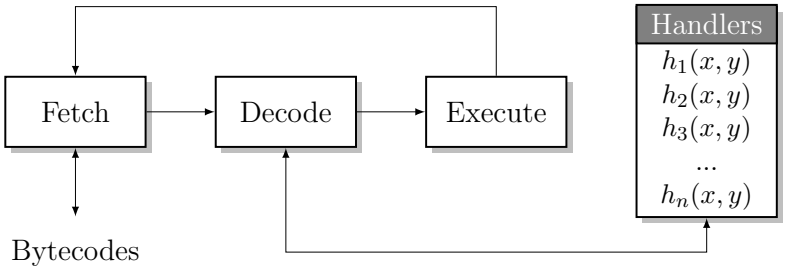


Figure 4.1: Virtualization-based obfuscation

4.2.2 Deobfuscation

Deobfuscation aims at reverting an obfuscated program back to a form close enough to the original one, or at least to a more understandable version. Along the previous years, *symbolic deobfuscation methods* based on advanced program analysis techniques have proven to be very efficient at breaking standard protections [133, 134, 135, 136, 30, 137, 31]. However, very effective countermeasures start to emerge, based on deep limitations of the underlying code-level reasoning mechanisms and potentially strongly limiting their usage [32, 33, 83, 146, 31]. Especially, all such methods are ultimately *sensitive to the syntactic complexity* of the code under analysis.

4.2.3 Search-based black-box deobfuscation

Search-based black-box deobfuscation has been recently proposed by Blazytko et al. [67], implemented in the SYNTIA tool, to learn the semantics of well-delimited code fragments, e.g. MBA expressions or VM handlers. The code under analysis is seen as a *black-box* that can only be queried (i.e., executed under chosen inputs to observe results). SYNTIA samples input-output (I/O) relations, then uses a learning engine to find an expression mapping sampled inputs to their observed outputs. Because it relies on a limited number of samples, results are not guaranteed to be correct. However, being fully black-box, it is in principle *insensitive to syntactic complexity.*

Scope. SYNTIA tries to infer a simple semantics of *heavily obfuscated local code fragments* – e.g., trigger based conditions or VM handlers. Understanding these fragments is critical to fulfill analysis.

Workflow. SYNTIA workflow is representative of search-based black-box deobfuscators. First, it needs (i) a *reverse window* i.e., a subset of code to work on; (ii) the location of its *inputs* and *outputs*. Consider the code in Listing 4.1 evaluating a condition at line 4. To understand this condition, a reverser focuses on the code between lines 1 and 3. This code segment is our reverse window. The reverser then needs to locate relevant inputs and outputs. The condition at line 4 is performed on $t3$. This is our output. The set of inputs contains any variables (registers or memory locations at assembly level) influencing the outputs. Here, inputs are x and y. Armed with these information, SYNTIA samples inputs randomly and observes resulting outputs. In our example, it might consider the samples $(x \mapsto 1, y \mapsto 2)$, $(x \mapsto 0, y \mapsto 1)$ and $(x \mapsto 3, y \mapsto 4)$ which respectively evaluate $t3$ to 3, 1 and 7. SYNTIA then synthesizes an expression matching these observed behaviors, using Monte Carlo Tree Search (MCTS) over the space of all possible (partial) expressions. Here, it rightly infers that $t3 \leftarrow x + y$ and the reverser concludes that the condition is $x + y = 5$, where a symbolic method will typically simply retrieve that $((x \vee 2y) \times 2 - (x \oplus 2y) - y) = 5$.

```
1 int t1 = 2 * y;
2 int t2 = x | t1;
3 int t3 = t2 * 2 - (x ^ t1) - y;
4 if (t3 == 5) ...
```

Listing 4.1: Obfuscated condition

4.3 Motivation

4.3.1 Attacker model

In the MATE scenario, the attacker is the software user himself. He has only access to the obfuscated version of the code under analysis and can read or run it at will. We consider that the attacker is highly skilled in reverse engineering but has limited resources in terms of time or money. We see reverse engineering as a *human-in-the-loop* process where the attacker combines manual analysis with automated state-of-the-art deobfuscation methods (slicing, symbolic execution, etc.) on critical, heavily obfuscated code fragments like VM handlers or trigger-based conditions. Thus, an effective defense strategy is to thwart automated deobfuscation methods.

4.3.2 Syntactic and semantic complexity

We now intuitively motivate the use of black-box deobfuscation. Consider that we reverse a piece of software protected through virtualization. We need to extract the semantics of all handlers, which usually perform basic operations like $h(x, y) = x + y$. Understanding h is trivial, but it can be protected to hinder analysis. Eq. (4.1) shows how MBA encoding hides h semantics.

$$h(x, y) = x + y \quad \overset{mba}{\longrightarrow} \quad (x \vee 2y) \times 2 - (x \oplus 2y) - y \tag{4.1}$$

Such encoding *syntactically* transforms the expression to make it incomprehensible while preserving its *semantics*. To highlight the difference between syntax and semantics, we distinguish:

1. **The syntactic complexity** of expression e is the size of e, i.e. the number of operators used in it;

2. **The semantic complexity** of expression e is the smallest size of expressions e' (in a given language) equivalent to e.

For example, in the MBA language, $x + y$ is syntactically simpler than $(x \vee 2y) \times 2 - (x \oplus 2y) - y$, yet they have the same semantic complexity as they are equivalent. Conversely, $x + y$ is more semantically complex than $(x + y) \wedge 0$, which equals 0. We do not claim to give a definitive definition of semantic

and syntactic complexity – as smaller is not always simpler – but introduce the idea that two kinds of complexity exist and are independent.

The encoding in Eq. (4.1) is simple, but it can be repeatedly applied to create a more syntactically complex expression, leading the reverser to either give up or try to simplify it automatically. White-box methods based on *symbolic execution* (SE) [30, 135] and *formula simplifications* (in the vein of compiler optimizations) can extract the semantics of an expression, yet they are sensitive to syntactic complexity and will not return simple versions of highly obfuscated expressions. Conversely, *black-box deobfuscation* treats the code as a black-box, considering only sampled I/O behaviors. *Thus increasing syntactic complexity, as usual state-of-the-art protections do, has simply no impact on black-box methods.*

4.3.3 Black-box deobfuscation in practice

We now present how black-box methods integrate in a global deobfuscation process and highlight crucial properties they must hold.

Global workflow. Reverse engineering can be fully automated, or handmade by a reverser, leveraging tools to automate specific tasks. While the deobfuscation process operates on the whole obfuscated binary, black-box modules can be used to analyze parts of the code like conditions or VM handlers. Upon meeting a complex code fragment, the black-box deobfuscator is called to retrieve a simple semantic expression. After synthesis succeeds, the inferred expression is used to help continue the analysis.

Requirements. In virtualization-based obfuscation, the black-box module is typically queried on all VM handlers [67]. As the number of handlers can be arbitrarily high, black-box methods need to be *fast*. In addition, inferred expressions should ideally be as *simple* as the original non-obfuscated expression and *semantically equivalent* to the obfuscated expression (i.e., correct). Finally, *robustness* (i.e., the capacity to synthesize complex expressions) is needed to be usable in various situations. Thus, **speed**, **simplicity**, **correctness** and **robustness**, are required for efficient black-box deobfuscation.

Discussion. One may argue that local black-box deobfuscation can be easily parallelized, limiting the need for fast synthesis. However, reverse engineering is often performed incrementally (e.g., packing, self-modification), or/and with a human in the loop and the need for quick feedback. In those scenarios, parallelization cannot help that much while slow synthesis obstructs analysis. Also, in some cases Syntia fails in 12h (Sections 4.5.3 and 4.8.2) – parallelism cannot help there.

4.4 Understand Black-box deobfuscation

We propose a general view of search-based code deobfuscation fitting state-of-the-art solutions [67, 73]. We also extend the evaluation of SYNTIA by Blazytko et al. [67], highlighting both some previously unreported weaknesses and strengths. From that we derive general lessons on the (in)adequacy of MCTS for code deobfuscation, that will guide our new approach (Section 4.5).

4.4.1 Problem at hand

Search-based deobfuscation takes an obfuscated expression and tries to infer an equivalent one with lower syntactic complexity. Such problem can be stated as following:

Deobfuscation. Let e, obf be 2 equivalent expressions such that obf is an obfuscated version of e – note that obf is possibly much larger than e. Deobfuscation aims to infer an expression e' equivalent to obf (and e), but with size similar to e. Such problem can be approached in three ways depending on the amount of information given to the analyzer:

Black-box We can only run obf. The search is thus driven by sampled I/O behaviors. SYNTIA [67] is a black-box approach;

Greybox Here obf is executable and readable but the semantics of its operators is mostly unknown. The search is driven by previously sampled I/O behaviors which can be applied to subparts of obf. QSYNTH [73] is a greybox solution;

White-box The analyzer has full access to obf (run, read) and the semantics of its operators is precisely known. Thus, the search can profit from advanced pattern matching and symbolic strategies. Standard static analysis falls in this category.

Black-box methods. Search-based black-box deobfuscators follow the framework given in Algorithm 2. In order to deobfuscate code, one must detail a *sampling strategy* (i.e., how inputs are generated), a *learning strategy* (i.e., how to learn an expression mapping sampled inputs to observed outputs) and a *simplification postprocess*. For example, **Syntia** samples inputs ***randomly***, uses ***Monte Carlo Tree Search*** (MCTS) [139] as learning strategy and leverages the ***Z3 SMT solver*** [150] for simplification. The choice of the sampling and learning strategies is critical. For example, too few samples could lead to incorrect results while too many could impact the search efficiency, and an inappropriate learning algorithm could impact robustness or speed.

Let us now discuss SYNTIA learning strategy. We show that using MCTS leads to disappointing performances and give insights to explain why.

Algorithm 2: Search-based black-box deobfuscation framework

In : The code to analyze *Code*; sampling strategy *Sample*; a learning strategy *Learn*; an expression simplifier $Simplify$;
Out : learned expression or Failure;

```
1 begin
2     Oracle ← Sample(Code)
3     success, expr ← Learn(Oracle)
4     if success then
5         return Simplify(expr)
6     else
7         return Failure
```

4.4.2 Evaluation of Syntia

We extend Syntia evaluation and tackle the following questions left unaddressed by Blazytko et al. [67].

RQ1 *Are results stable across different runs?*
This is desirable due to the stochastic nature of MCTS;

RQ2 *Is* Syntia *fast, robust and does it infer simple and correct results?*
Syntia offers *a priori* no guarantee of correctness nor quality. Also, we consider small time budget (1s), adapted to human-in-the-loop scenarios but absent from the initial evaluation;

RQ3 *How is synthesis impacted by the set of operators size?*
Syntia learns expressions over a search space fixed by predefined grammars. Intuitively, the more operators in the grammar, the harder it will be to converge to a solution. We use 3 sets of operators to assess this impact.

4.4.2.1 Experimental setup

We distinguish the **success rate** (number of expressions inferred) from the **equivalence rate** (number of expressions inferred and equivalent to the original one). The equivalence rate relies on the Z3 SMT solver [150] with a timeout of 10s. Since Z3 timeouts are inconclusive answers, we define a notion of **equivalence range**: its lower bound is the **proven equivalence rate** (number of expressions proven to be equivalent) while its upper bound is the **optimistic equivalence rate** (expressions not proven different, i.e., optimistic = proven + #timeout). The equivalence rate is within the equivalence range, while the success rate is higher than the optimistic equivalence rate. Finally, we define the **quality** of an expression as the ratio between the number of operators in recovered and target expressions. It estimates the syntactic complexity of inferred expressions compared to the original ones. A

quality of 1 indicates a perfect result: the recovered expression has the same size as the target expression.

Benchmarks. We consider two benchmark suites: B1 and B2. B1[1] comes from Blazytko et al. [67] and was used to evaluate SYNTIA. It comprises 500 randomly generated expressions with up to 3 arguments, and simple semantics. It aims at representing state-of-the-art VM-based obfuscators. *However, we found that B1 suffers from several significant issues*: (1) it is not well distributed over the number of inputs and expression types, making it unsuitable for fine-grained analysis; (2) only 216 expressions are unique modulo renaming – the other 284 expressions are α-equivalent, like x+y and a+b. These problems threaten the validity of the evaluation.

We thus *propose a new benchmark* **B2** consisting of 1,110 randomly generated expressions, better distributed according to the number of inputs and the nature of operators – see Table 4.1. We use three categories of expressions: Boolean, Arithmetic and Mixed Boolean-Arithmetic, with 2 to 6 inputs. Especially, expressions are spread equally between categories to prevent biased results. Each expression has an Abstract Syntax Tree (AST) of maximal height 3. As a result, B2 is more challenging than B1 and enables a finer-grained evaluation. Considering such diverse and complex expressions is crucial as black-box deobfuscation evolves in an adversarial context where limitations can be exploited to thwart analysis.

Note that we also consider **QSynth data-sets** [73] in Section 4.6, developed by the Quarkslab R&D company.

	Type			# Inputs				
	Bool.	Arith.	MBA	2	3	4	5	6
#Expr.	370	370	370	150	600	180	90	90

Table 4.1: Distribution of samples in benchmark B2

Operator sets. Table 4.2 introduces three operator sets: FULL, EXPR and MBA. We use these to evaluate sensitivity to the search space and answer **RQ3**. EXPR is as expressive as FULL even if EXPR $\subset$ FULL. MBA can only express Mixed Boolean-Arithmetic expressions [138].

Table 4.2: Sets of operators

Full : $\{-_1, \neg, +, -, \times, \gg_u, \gg_s, \ll, \wedge, \vee, \oplus, \div_s, \div_u, \%_s, \%_u, ++ \}$
Expr : $\{-_1, \neg, +, -, \times, \wedge, \vee, \oplus, \div_s, \div_u, ++ \}$
Mba : $\{-_1, \neg, +, -, \times, \wedge, \vee, \oplus\}$

[1] https://github.com/RUB-SysSec/syntia/tree/master/samples/mba/tigress

Configuration. We run all our experiments on a machine with 6 Intel Xeon E-2176M CPUs and 32 GB of RAM. We evaluate SYNTIA in its original configuration [67]: the SA-UCT parameter is 1.5, we use 50 I/O samples and a maximum playout depth of 0. We also limit SYNTIA to 50,000 iterations per sample, corresponding to a timeout of 60s per sample on our test machine.

4.4.2.2 Evaluation Results

Let us summarize here the outcome of our experiments.

RQ1. Over 15 runs, SYNTIA finds between 362 and 376 expressions of B1 i.e., 14 expressions of difference (2.8% of B1). Over B2, it finds between 349 and 383 expressions i.e., 34 expressions of difference (3.06% of B2). Hence, SYNTIA *is very stable across executions.*

RQ2. SYNTIA cannot efficiently infer B2 ($\approx$ 34% success rate). Moreover, Table 4.3 shows SYNTIA to be highly sensitive to time budget. More precisely, with a time budget of 1 s/expr., SYNTIA only retrieves 16.3% of B2. Still, even with a timeout of 600 s/expr., it tops at 42% of B2. In addition, SYNTIA is unable to synthesize expressions with more than 3 inputs – success rates for 4, 5 and 6 inputs respectively falls to 10%, 2.2% and 1.1%. It also struggles over expressions using a mix of Boolean and arithmetic operators, synthesizing only 21% (see Table 4.9). Still, SYNTIA performs well regarding quality and correctness. On average, its quality is around 0.60 (for a timeout of 60 s/expr.) i.e., resulting expressions are simpler than the original (non obfuscated) ones, and it rarely returns non-equivalent expressions – between 0.5% and 0.8% of B2. We thus conclude that SYNTIA *is stable and returns correct and simple results. Yet, it is not efficient enough (solves only few expressions on B2, heavily impacted by time budget) and not robust (number of inputs and expression type).*

Table 4.3: SYNTIA depending on the timeout per expression (B2)

	1s	10s	60s	600s
Succ. Rate	16.5%	25.6%	34.5%	42.3%
Equiv. Range	16.3%	25.1 - 25.3%	33.7 - 34.0%	41.4 - 41.6%
Mean Qual.	0.35	0.49	0.59	0.67

RQ3. Default SYNTIA synthesizes expressions over the FULL set of operators. To evaluate its sensitivity to the search space we run it over FULL, EXPR and MBA. Smaller sets do exhibit higher success rates (42% on MBA) but results remain disappointing. SYNTIA *is sensitive to the size of the operator set but is inefficient even with* MBA.

Conclusion. Syntia *is stable, correct and returns simple results. Yet, it is heavily impacted by the time budget and lacks robustness. It thus fails to meet the requirements given in Section 4.3.3.*

4.4.3 Optimal Syntia

To ensure the conclusions given in Section 4.4.4 apply to MCTS and not only to Syntia, we study Syntia extensively to find better set ups for the following parameters: simulation depth, SA-UCT value (configuring the balance between exploitative and explorative behaviors), number of I/O samples and distance.

Table 4.4: Syntia depending on max playout depth (Mba, B2, timeout=60s).

Max play. depth	0	3	5
Succ. Rate	42.6 %	31.8 %	28.6 %
Equiv. Range	42.3 - 42.6 %	31.4 - 31.8 %	28.1 - 28.6 %
Mean Qual.	0.66	1.03	1.06

Simulation depth. As presented in Section 4.4.4, MCTS simulates each generated nodes. To do so, it applies rules of the grammar randomly to the non terminal expression until it becomes terminal. An important parameter is thus the maximum simulation depth i.e., the number of rules not leading to terminal nodes (like $U \rightarrow U + U$). By default, Syntia considers a maximum simulation depth of 0, which mean that all non terminal symbols are directly replaced by variables or constant values. Table 4.4 shows that increasing this parameter is not beneficial.

Number of I/O samples. By defaults Syntia considers 50 samples. Table 4.5 presents results for different number of samples. We observe little improvement when the number of samples decreases. Still, it stays in the same range of results.

Table 4.5: Syntia for different number of samples (B2, Mba, timeout=60s).

# samples	10	20	50	100
Succ. Rate	45.6%	44.9%	42.6%	43.2%
Equiv. Range	45.1 - 45.4%	44.7 - 44.9%	42.3% - 42.6%	42.9 - 43.2%
Mean Qual.	0.69	0.71	0.66	0.69

Objective function. By default, Syntia evaluates if an expression is close to the target one by computing the mean between different distances. To complete our evaluation of Syntia we launched it with Xyntia Log-arithmetic

distance. We observe that as XYNTIA the log-arithmetic seems more appropriate to guide the search. Still, SYNTIA success rate stays bellow 50%.

Table 4.6: SYNTIA depending on the objective function (B2, MBA, timeout=60s).

	SYNTIA-dist	Log-arith
Succ. Rate	42.6%	47.9%
Equiv. Range	42.3 - 42.6%	47.4 - 47.9%
Mean Qual.	0.66	0.70

Simulated annealing UCT (SA-UCT). From a high level, MCTS can be divided in 2 behaviors: exploitation (where it focuses on promising nodes) and exploration (where it checks rarely visited or at first glance non interesting nodes). The SA-UCT constant value is a parameter to configure the balance between these behaviors. The smaller is the constant value the more exploitative MCTS is. On the contrary, the bigger it is, more explorative is MCTS. By default SYNTIA sets the SA-UCT constant value to 1.5. Table 4.7 presents results of SYNTIA for smaller and bigger values. For smaller values, SYNTIA is less efficient. This is coherent with claims from Section 4.4.4. Indeed, as the search space is highly unstable, simulations are misleading. Thus, focusing too much on exploitation is unsuitable. However, it also appears that, bigger values can be beneficial. This is also coherent with Section 4.4.4 as it shows that the most important behavior is exploration. Still, even with SA-UCT values > 1.5 success rate stays low ($< 50\%$).

Table 4.7: SYNTIA depending on SA-UCT value (MBA, B2, timeout = 60 s).

SA-UCT	3	2	1.5	0.5	0.1
Succ. Rate	48.0%	48.2%	42.6 %	34.6 %	19.1 %
Equiv. Range	47.7 - 48.0%	48.1 - 48.2 %	42.3 - 42.6 %	34.6 %	19.1 %
Mean Qual.	0.71	0.72	0.66	0.62	0.44

Optimal Syntia. Our extensive study highlights a new optimal configuration of SYNTIA (MBA set of operators, simulation depth=0, #samples=10, objective function=log-arithmetic, SA-UCT=2). However, even with this configuration, SYNTIA success rate stays around 50% (Table 4.8). While slightly better, such results are still disappointing.

Conclusion. *By default,* SYNTIA *is well configured. Changing its parameters lead in the best scenario to marginal improvement, hence the pitfalls highlighted seem to be inherent to the MCTS approach.*

Table 4.8: Optimal SYNTIA (B2, timeout = 60 s).

Succ. Rate	52.7%
Equiv. Range	52.1 - 52.6%
Mean Qual.	0.76

4.4.4 MCTS for deobfuscation

Let us explore whether these issues are related to MCTS.

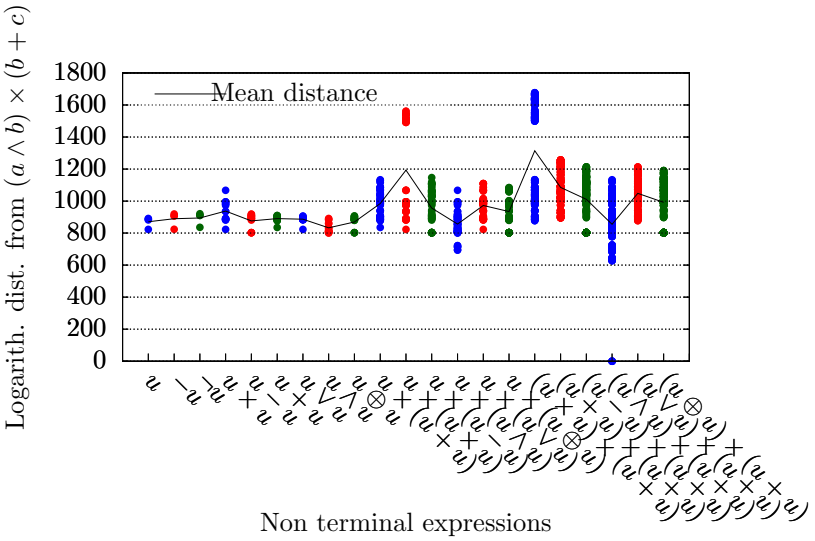
Monte Carlo Tree Search. MCTS creates here a search tree where each node is an *expression* which can be *terminal* (e.g. $a + 1$, where a is a variable) or *partial* (e.g. $U + a$, where U is a non-terminal symbol). The goal of MCTS is to expand the search tree smartly, *focusing on most pertinent nodes first*. Evaluating the pertinence of a *terminal node* is done by *sampling* (computing here a distance between the evaluation of sampled inputs over the node expression against their expected output values). For *partial nodes*, MCTS relies on *simulation*: random rules of the grammar are applied to the expression (e.g., $U + a \rightsquigarrow b + a$) until it becomes terminal and is evaluated. As an example, let $\{(a \mapsto 1, b \mapsto 0), (a \mapsto 0, b \mapsto 1)\}$ be the sampled inputs. The expression $b + a$ (simulated from $U + a$) evaluates them to $(1, 1)$. If the ground-truth outputs are 1 and -1, the distance will equal $\delta(1, 1) + \delta(1, -1)$ where δ is a chosen distance function. We call the result the *pertinence measure*. The closer it is to 0, the more pertinent the node $U + a$ is considered and the more the search will focus on it.

Analysis. This *simulation-based pertinence estimation* is not reliable in our code deobfuscation setting.

- We present in Fig. 4.2, for different non-terminal nodes, the distance values computed through simulations. We observe that from a starting node, a random simulation can return drastically different results. It shows that *the search space is very unstable* and that relying on simulation is misleading (especially in our context where time budget is small);

- Moreover, our experiments show that in practice SYNTIA is not guided by simulations and behaves *almost as if it were an enumerative (BFS) search* – MCTS where simulations are non informative. As an example, Fig. 4.3 compares how the distance evolves over time for SYNTIA and a custom, fully enumerative, MCTS synthesizer: both are very similar. Actually, SYNTIA and enumerative MCTS perform similarly over B2: with a 60s (resp. 600s) timeout, enumerative MCTS reaches 41.4% (resp. 51.6%) success rate vs. 42.6% (resp. 54.9%) for Syntia (MBA operators set);

- Finally, on B2 (resp. B1) with a timeout of 60s, only 34/341 (resp. 20/376) successfully synthesized expressions are the children of previously most promising nodes. It shows that SYNTIA successfully synthesized expressions due to its exploratory (i.e., enumerative) behavior rather than to the selection of nodes according to their *pertinence*.



Each point represents the distance between $(a \wedge b) \times (b + c)$ and one simulation of a non terminal expression (horizontal axis). A non terminal expression, can generate multiple terminal ones through simulations, leading to completely different results.

Figure 4.2: Dispersion of the distance for different simulations

Conclusion. *The search space from black-box code deobfuscation is too unstable, making MCTS simulations unreliable. MCTS in that setting is then almost enumerative and inefficient. That is why* SYNTIA *is slow and not robust, but returns simple expressions.*

4.4.5 Conclusion

While SYNTIA returns simple results, it only synthesizes semantically simple expressions and is slow. These unsatisfactory results can be explained by the fact that the search space is too unstable, making the use of MCTS unsuitable. In the next section, we show that methods avoiding the manipulation of partial expressions (and thus free from simulation) are better suited to deobfuscation.

4.5 Improve Black-box deobfuscation

We define a new search-based black-box deobfuscator, dubbed XYNTIA, leveraging *S-metaheuristics* [144] and *Iterated Local Search* (ILS) [145] and com-

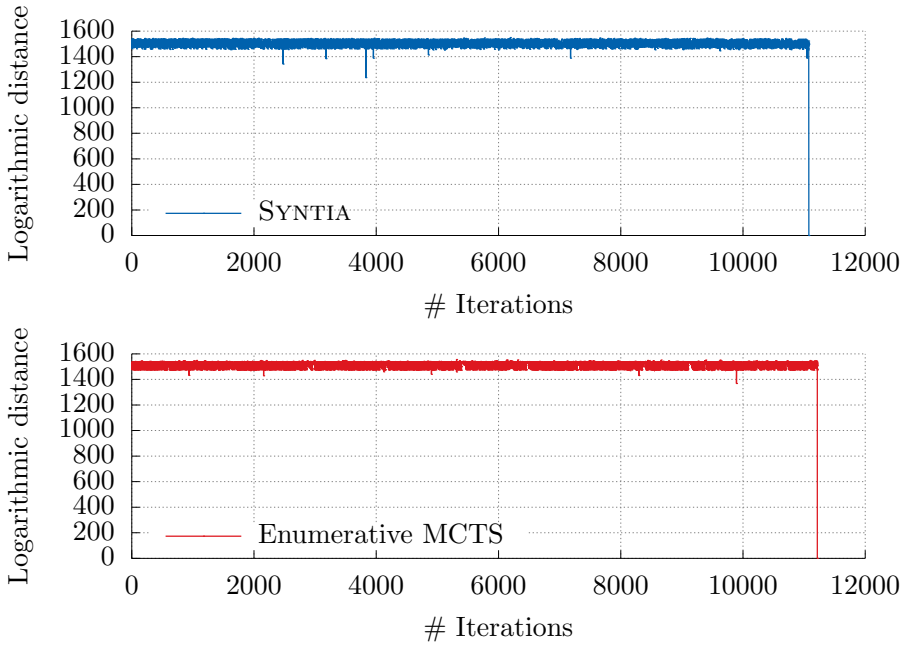


Figure 4.3: SYNTIA and enumerative MCTS distance evolution (expression successfully synthesized)

pare its design to rival deobfuscators. Unlike MCTS, S-metaheuristics *only manipulate terminal expressions* and do not create tree searches, thus we expect them to be better suited than MCTS for code deobfuscation. Among S-metaheuristics, ILS is particularly *designed for unstable search spaces*, with the ability to remember the last best solution encountered and to restart the search from that point. We show that these methods are well-guided by the distance function and significantly outperform MCTS in the context of black-box code deobfuscation.

4.5.1 Deobfuscation as optimization

As presented in Section 4.4, SYNTIA frames deobfuscation as a single player game. We instead propose to frame it as an optimization problem using ILS as learning strategy.

Black-box deobfuscation: an optimization problem. Black-box deobfuscation synthesizes an expression from input-output samples and can be modeled as an optimization problem. The objective function, noted f, measures the similarity between current and ground truth behaviors by computing the sum of the distances between found and objective outputs. The goal is to infer an expression minimizing the objective function over the I/O samples. If the underlying grammar is expressive enough, a minimum exists and matches all sampled inputs to objective outputs, zeroing f. The reliability of the found

solution depends on the number of I/O samples considered. Too few samples would not restrain search enough and lead to flawed results.

Solving through search heuristics. S-metaheuristics [144] can be advantageously used to solve such optimization problems. A wide range of heuristics exists (Hill Climbing, Random Walk, Simulated Annealing, etc.). They all iteratively improve a candidate solution by testing its "neighbors" and moving along the search space. Because solution improvement is evaluated by the objective function, it is said to guide the search.

Iterated Local Search. Some S-metaheuristics are prone to be stuck in local optimums so that the result depends on the initial input chosen. Iterated Local Search (ILS) [145] tackles the problem through iteration of search and the ability to restart from previously seen best solutions. Note that ILS is parameterized by another search heuristics (for us: Hill Climbing). Once a local optimum is found by this side search, ILS perturbs it and uses the perturbed solution as initial state for the side search. At each iteration, ILS also saves the best solution found. Unlike most other S-metaheuristics (Hill Climbing, Random Walk, Metropolis Hasting and Simulated Annealing, etc.), if the search follows a misleading path, ILS can restore the best seen solution so far to restart from an healthy state.

4.5.2 Xyntia internals

Xyntia is built upon 3 components: the *optimization* problem we aim to solve, the *oracle* which extracts the sampling information from the protected code under analysis and the *search heuristics*.

Oracle. The *oracle* is defined by the sampling strategy which depicts how the protected program must be sampled and how many samples are considered. As default, we consider that our oracle samples 100 inputs over the range $[-50; 49]$. Five are not randomly generated but equal interesting constant vectors $(\vec{0}, \vec{1}, \vec{-1}, \vec{min}_s, \vec{max}_s)$. These choices arise from a systematic study of the different settings to find the best design (see Section 4.5.4).

Optimization problem. The *optimization problem* is defined as follow. The search space is the set of expressions expressible using the Expr set of operators (see Table 4.2), and considers a unique constant value `1`. This grammar enables Xyntia to reach optimal results while being as expressive as Syntia [67]. Besides, we consider the objective function:

$$f_{\vec{o}^*}(\vec{o}) = \sum_i log_2(1 + |o_i - o_i^*|)$$

It computes the Log-arithmetic distance between synthesized expressions outputs ($\vec{o}$) and sampled ones ($\vec{o}^*$). The choice of the grammar and of the objective function are respectively discussed in Sections 4.5.3 and 4.5.4.

Search. XYNTIA leverages Iterated Local Search (ILS) to minimize our objective function and so to synthesize target expressions. We present now how ILS is adapted to our context. ILS applies two steps starting from a random terminal (constant value or variable):

- ILS reuses the *best expression found so far* to *perturb it* by randomly selecting a node of the AST and replacing it by a random *terminal* node. The resulting AST is kept even if the distance increases and passed to the next step;
- *Iterative Random Mutations:* the side search (in our case Hill Climbing) iteratively mutates the input expression until it cannot improve anymore. We estimate that no more improvement can be done after 100 inconclusive mutations. A mutation consists in replacing a randomly chosen node of the abstract syntax tree (AST) by a leaf or an AST of depth one (only one operator) – e.g. $\boxed{1} + (-a) \rightsquigarrow (-b) + (-a)$. At each mutation, it keeps the version of the AST minimizing the distance function. During mutations, the *best solution so far* is updated to be restored in the perturbation step. If a solution nullifies the objective function, it is directly returned.

These two operations are iteratively performed until time is out (by default **60s**) or an expression mapping all I/O samples is found. Furthermore, as SYNTIA applies Z3 simplifier to "clean up" recovered expressions, we add a custom *post-process expression simplifier*, applying simple rewrite rules until a fixpoint is reached. It significantly improves the quality of the expressions while adding no significant overhead (+2.6ms on average). XYNTIA is implemented in *OCaml* [151], within the BINSEC framework for binary-level program analysis [25]. It comprises ≈9k lines of code.

4.5.3 Xyntia evaluation

We now evaluate XYNTIA in depth and compare it to SYNTIA. As with SYNTIA we answer the following questions:

RQ4 *Are results stable across different runs?*
RQ5 *Is* XYNTIA *robust, fast and does it infer simple and correct results?*
RQ6 *How is synthesis impacted by the set of operators size?*

Configuration. For all our experiments, we default to locally optimal XYNTIA ($\text{XYNTIA}_{\text{OPT}}$) presented in Section 4.5.2. It learns expressions over EXPR, samples 100 inputs (95 randomly and 5 constant vectors) and uses the Logarithmetic distance as objective function.

Interestingly, all results reported here also hold (to a lesser extend regarding efficiency) for other XYNTIA *configurations (Section 4.5.4), especially these versions consistently beat* SYNTIA.

RQ4. Over 15 runs XYNTIA always finds all 500 expressions in B1 and between 1051 and 1061 in B2. Thus, XYNTIA *is very stable across executions.*

RQ5. Unlike SYNTIA, XYNTIA performs very well on both B1 and B2 with a timeout of 60 s/expr. Fig. 4.4 reveals that it is still successful for a timeout of 1 s/expr. (78% proven equivalence rate), where it finds 2× more expressions than SYNTIA with a timeout of 600 s/expr.

We also observe such tendency over B1 and BP1 (see Section 4.8.2) and for 12h timeout. On B1, SYNTIA reaches 41%, 74%, 88.2% and 97.6% success rate for respectively 1s, 60s, 600s and 12h timeout, against 100% success rate for XYNTIA in 1s. For BP1, SYNTIA finds only 1/15 expressions with a 12h timeout against 12/15 for XYNTIA in 60s. From evaluation on B1 and B2, it appears that SYNTIA success rate increases logarithmically over time. Thus, time budget needed for SYNTIA to catch XYNTIA is expected to be unrealistic.

In addition, XYNTIA handles well expressions using up to 5 arguments and all expression types (Table 4.9). Its mean quality is around 0.93, which is very good (objective is 1), and it rarely returns not equivalent expressions – only between 1.3% and 4.9%. Thus, XYNTIA *reaches high success and equivalence rate. It is fast, synthesizing most expressions in* $\leq 1s$*, and it returns simple and correct results.*

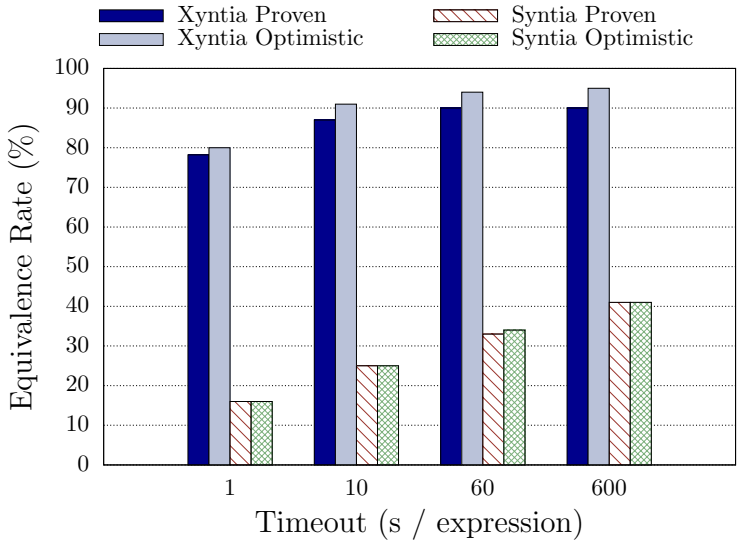


Figure 4.4: Equivalence range of SYNTIA and XYNTIA ($\text{XYNTIA}_{\text{OPT}}$) depending on timeout (B2)

RQ6. XYNTIA by default synthesizes expressions over EXPR while SYNTIA infers expressions over FULL. To compare their sensitivity to search space and show that previous results are not due to search space inconsistency, we run both tools over FULL, EXPR and MBA. Experiments show that XYNTIA reaches high equivalence rates for all operators sets while SYNTIA results stay

		Bool.	Arith.	MBA
Syntia	Succ. Rate	53.8%	28.6%	21.1%
	Equiv. Range	53.0%	27.8 - 28.1%	20.3 - 20.8%
	Mean Qual.	0.53	0.61	0.71
Xyntia	Succ. Rate	98.4%	96.5%	91.6%
	Equiv. Range	97.8%	88.9 - 94.9%	85.1 - 90.0%
	Mean Qual.	0.73	1.0	1.05

Table 4.9: Syntia & Xyntia ($\text{Xyntia}_{\text{Opt}}$): results according to expression type (B2, timeout = 60 s)

low. Still, Xyntia seems more sensitive to the size of the set of operators than Syntia. Its proven equivalence rate decreases from 90% (Expr) to 71% (Full) while Syntia decreases only from 38.7% (Expr) to 33.7% (Full). Conversely, as for Syntia, restricting to Mba benefits to Xyntia (proven equiv. rate: 91%). Thus, *like* Syntia*,* Xyntia *is sensitive to the size of the operators set. Yet,* Xyntia *reaches high equivalence rates even on* Full *while* Syntia *remains inefficient even on* Mba.

Conclusion. Xyntia *is a lot faster and more robust than* Syntia*. It is also stable and returns simple expressions. Thus,* Xyntia*, unlike* Syntia*, meets the requirements given in Section 4.3.3.*

4.5.4 Optimal Xyntia and other S-Metaheuristics

Previous experiments consider the $\text{Xyntia}_{\text{Opt}}$ configuration of Xyntia. It comes from a systematic evaluation of the design space. To do so, we considered (i) different S-metaheuristics: Hill Climbing (HC), Random Walk (RW), Simulated Annealing (SA), Metropolis Hasting (MH) and Iterated Local Search (ILS); (ii) different sampling strategies; (iii) different objective functions. This evaluation confirms that $\text{Xyntia}_{\text{Opt}}$ is locally optimal and that ILS, being able to restore best expression seen after a number of unsuccessful mutations, outperforms other S-metaheuristics (Table 4.10). Moreover, all S-metaheuristics – except Hill Climbing – outperforms Syntia.

Table 4.10: Synthesis Equivalence Rate for different S-metaheuristics (B2, $\text{Xyntia}_{\text{Opt}}$, timeout = 60 s)

	RW	HC	ILS	SA	MH
Equiv. Range	62.3 - 63.4%	31.9 - 33.1%	**90.6 - 94.2%**	64.8 - 65.8%	57.7 - 58.5%

Conclusion. *Principled and systematic evaluation of* Xyntia *design space leads to the locally optimal* $\text{Xyntia}_{\text{Opt}}$ *configuration. It notably shows that*

ILS outperforms other tested S-metaheuristics. Moreover, all these S-metaheuristics – except Hill Climbing – outperform MCTS, confirming that manipulating only terminal expressions is beneficial.

4.5.5 On the effectiveness of ILS over MCTS

We present in Fig. 4.5 the typical distance evolution along the search process when using XYNTIA. We can see that the distance follows a step-wise progression, which is drastically different from the case of SYNTIA and enumerative MCTS (Fig. 4.3). Hence, unlike them, XYNTIA is indeed guided by the distance function. Moreover, note that XYNTIA globally follows a positive trend i.e., it does not unlearn previous work. Indeed, before each perturbation, the best expression found from now is restored. Thus, if iterative mutations follows a misleading path, the resulting solution is not kept and the best solution is reused to be perturbed. Keeping the current best solution is of first relevance as the search space is highly unstable and enables XYNTIA to be more reliable and less dependant of randomness.

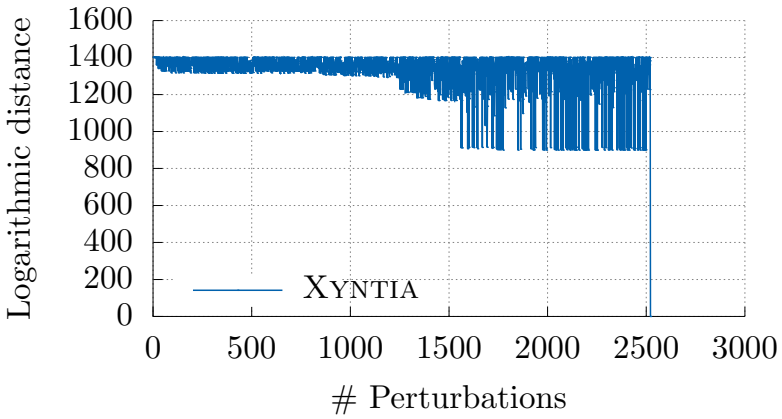


Figure 4.5: XYNTIA (XYNTIA$_{\text{OPT}}$) distance evolution (expression successfully synthesized)

Conclusion. *Unlike MCTS, which is almost enumerative in code deobfuscation, ILS is well guided by the objective function and distance evolution follows a positive trend. This is true as well for other S-metaheuristics.*

4.5.6 Conclusion

We resume here obtained results and present limitations of black-box approaches.

4.5.6.1 Conclusion

Because of the high instability of the search space, *Iterated Local Search* is much more appropriate than MCTS (and, to a lesser extent, than other S-metaheuristics) for black-box code deobfuscation, as it manipulates terminal expressions only and is able to restore the best solution seen so far in case the search gets lost. These features enable Xyntia to keep the advantages of Syntia (stability, output quality) while clearly improving over its weaknesses: especially Xyntia manages with 1s timeout to synthesize twice more expressions than Syntia with 10min timeout.

Other S-metaheuristics also perform significantly better than MCTS here, demonstrating that the problem itself is not well-suited for partial solution exploration and simulation-guided search.

4.5.6.2 Limitations

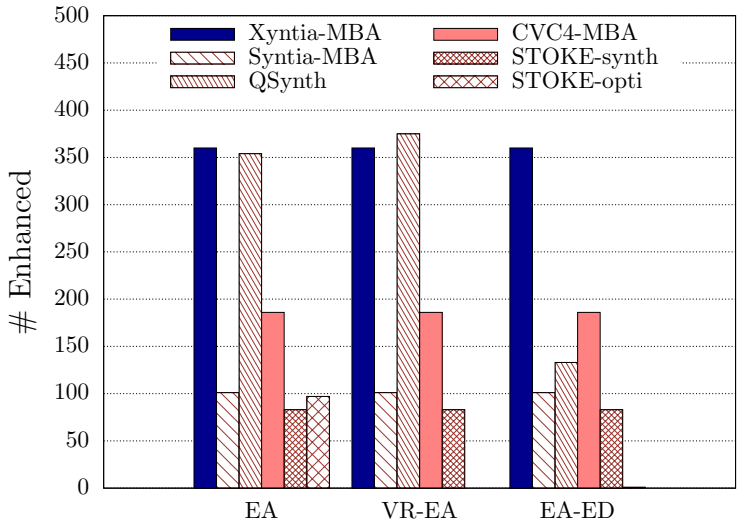
Black-box approaches must consider limited languages to be efficient. This restricts their use to local contexts – e.g., analyzing sets of code blocks rather than full modules.

Moreover, synthesis relies on two main steps, sampling and learning, which both show weaknesses. Indeed, Xyntia and Syntia randomly sample inputs to approximate the semantics of an expression. It then assumes that samples depict all behaviors of the code under analysis. If this assumption is invalid then the learning phase will miss some behaviors, returning partial results. As such, black-box deobfuscation is unsuitable to handle point functions.

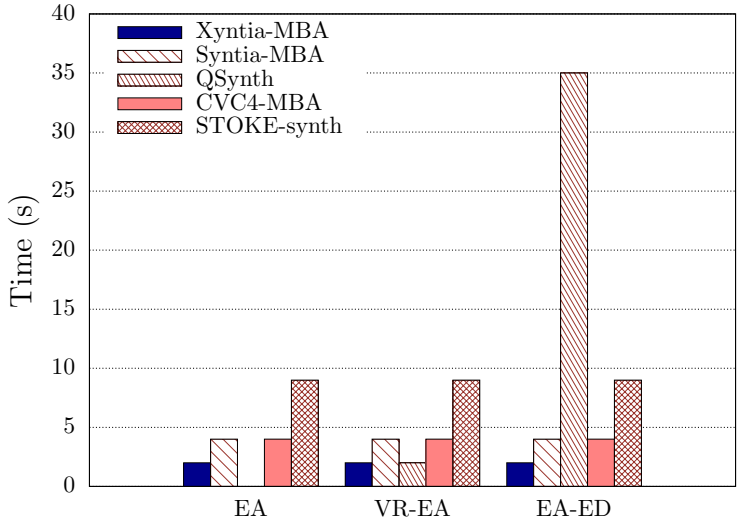
Learning can itself be impacted by other factors. For instance, *semantically complex expressions* are hard to infer. While they are rare in local code, we show in Section 4.8 how to take advantage of them to protect against black-box attacks. A related problem are expressions with unexpected constant values. They are hard to handle as the grammar of Xyntia and Syntia only considers the constant value `1`. Thus, finding expressions with constant values absent from the grammar requires to create them (e.g., encoding 3 as 1+1+1), which may be unlikely. A naive solution is to add to the grammar additional constant values but it significantly impacts efficiency. Indeed, for 100 values [0; 99]), the equivalence rate is divided by 2 (resp., by 4 for 200 values). Still, Section 4.7 shows that Xyntia handles usual interesting constant values (unlike Syntia).

4.6 Comparison with other approaches

We now extend the comparison to other state-of-the-art tools: (i) a greybox deobfuscator (QSynth [73]); (ii) white-box simplifiers (GCC, Z3 simplifier and our custom simplifier); (iii) program synthesizers (CVC4 [91], winner of the SyGus'19 syntax-guided synthesis competition [152] and STOKE [10], an



(a) Enhancement rate



(b) Mean synthesis time per expression – STOKE-opti not shown as it always uses 60 s

Figure 4.6: Syntia, QSynth, Xyntia, CVC4 and STOKE on EA, VR-EA and EA-ED data-sets (timeout = 60 s)

efficient superoptimizer). Unlike black-box approaches, greybox and white-box methods should be evaluated on the enhancement rate, as otherwise they can always succeed by returning the obfuscated expression. The enhancement rate measures how often synthesized expressions are smaller than the *original* ones ($quality \leq 1$).

Benchmarks. We compare black-box program synthesizers on B2, and grey/white

box approaches on the three QSynth data-sets,[2] each of them comprising 500 expressions obfuscated with Tigress [142]: **EA** (base data-set, obfuscated with the *EncodeArithmetic* transformation), **VR-EA** (EA obfuscated with *Virtualize* and *EncodeArithmetic* protections), and **EA-ED** (EA obfuscated with *EncodeArithmetic* and *EncodeData* transformations).

White-box. We first compare Xyntia over the EA, VR-EA and EA-ED data-sets with 3 white-box approaches: GCC, Z3 simplifier (v4.8.7) and our custom simplifier. As expected, they are not efficient compared to Xyntia. Regardless of the data-set, they simplify ≤ 68 expressions where Xyntia simplifies 360 of them.

Greybox. We now compare Xyntia to QSynth published results [73] on EA, VR-EA and EA-ED. Fig. 4.6a shows that while both tools reach comparable results (enhancement rate $\approx 350/500$) for simple obfuscations (EA and VR-EA), Xyntia keeps the same results for heavy obfuscations (EA-ED) while QSynth drops to 133/500. Actually, unlike QSynth, Xyntia is insensitive to syntactic complexity.

Program synthesizers. We finally compare Xyntia to state-of-the-art program synthesizers, namely CVC4 [91] and STOKE [10]. CVC4 takes as input a grammar and a specification and returns, through enumerative search, a consistent expression. STOKE is a super-optimizer leveraging program synthesis (based on Metropolis Hasting) to infer optimized code snippets. It does not return an expression but optimized assembly code. STOKE addresses the optimization problem in two ways: (i) STOKE-synth starts from a pre-defined number of nops and mutates them; (ii) STOKE-opti starts from the non-optimized code and mutates it to simplify it. While STOKE integrates its own sampling strategy and grammar, CVC4 does not – thus, we consider for CVC4 the same sampling strategy as Xyntia (100 I/O samples with 5 constant vectors) as well as the Expr and Mba grammars. More precisely, CVC4-Expr is used over B2 to compare to Xyntia ($\text{Xyntia}_{\text{Opt}}$) and CVC4-Mba is evaluated on EA, VR-EA and EA-ED to compare against QSynth.

Our experiments show that CVC4-Expr and STOKE-synth synthesize less than 40% of B2 (respectively 36.8% and 38.0%) while Xyntia reaches 90.6% proven equivalence rate. Indeed enumerative search (CVC4) is less appropriate when time is limited. Results of STOKE-synth are also expected as its search space considers all assembly mnemonics. Moreover, Fig. 4.6a shows that black-box and white-box (STOKE-opti) synthesizers do not efficiently simplify obfuscated expressions. STOKE-opti finds only 1 / 500 expressions over EA-ED and does not handle jump instructions, inserted by the VM, failing to analyze VR-EA.

Conclusion. Xyntia rivals QSynth on light / mild protections and outperforms it on heavy protections, while pure white-box approaches are far behind,

[2] https://github.com/werew/qsynth-artifacts

showing the benefits of being independent from syntactic complexity. Also, XYNTIA outperforms state-of-the-art program synthesizers showing that it is better suited to perform deobfuscation. These good results show that seeing deobfuscation as an optimization problem is fruitful.

4.7 Deobfuscation with Xyntia

We now prove that XYNTIA is insensitive to common protections (opaque predicates) as well as to recent anti-analysis protections (MBA, covert channels, path explosion) and we confirm that black-box methods can help reverse state-of-the-art virtualization [142, 140].

4.7.1 Effectiveness against usual protections

XYNTIA is able to bypass many protections (see Table 4.11):

Mixed Boolean-Arithmetic [138] hides the original semantics of an expression both to humans and SMT solvers. However, the encoded expression remains equivalent to the original one. As such, the semantic complexity stays unchanged, and XYNTIA should not be impacted. Launching XYNTIA on B2 obfuscated with Tigress [142] *Encode Arithmetic* transformation (size of expression: x800) confirms that it has no impact: equivalence range with and without protection respectively equals 90.0 - 93.8% and 90.6 - 94.2%.

Opaque predicates [12] obfuscate control flow by creating artificial conditions in programs. The conditions are traditionally tautologies and dynamic runs of the code will follow a unique path. Thus, sampling is not affected and synthesis not impacted. We show it by launching XYNTIA over B2 obfuscated with Tigress *AddOpaque* transformation (result: equiv. range is 89.9 - 93.0%).

Path-based obfuscation [146, 32] takes advantage of path explosion to thwart symbolic execution, massively adding additional feasible paths. While it is efficient against symbolic-based analysis, what about black-box ones? To evaluate its impact, we encoded expressions as in Listing 4.2. This example is inspired by the **For** primitive from [32]. It computes the sum of `x` and `y` adding loops to increase the number of paths to explore (one path for each value of x and y), effectively killing symbolic execution. However, black-box deobfuscation sees input-output behaviors only and successfully bypass this protection: equivalence range with and without protections respectively equals 90.6 - 94.2% and 89.5 - 93.7%.

```
1 int sum(int x, int y){
2     int x1, y1;
3     for (int i = 0; i < x; i++) x1++;
4     for (int i = 0; i < y; i++) y1++;
5     return x1 + y1;
6 }
```

Listing 4.2: Sum function with path-oriented obfuscation

Covert channels [83] hide information flow to static analyzers by rerouting data to invisible part of the states (usually OS related) before retrieving it, for example taking advantage of timing difference between a slow thread and a fast thread. Again, as black-box deobfuscation focuses only on input-output relationships, covert channels should not disturb it. Note that the probabilistic nature of such obfuscations (obfuscated behaviours can differ from unobfuscated ones from time to time) could be a problem in case of high fault probabilities, but in order for the technique to be useful, fault probability must precisely remains low. We show it has no impact by obfuscating B2 with the *InitEntropy* and *InitImplicitFlow* (thread kind) transformations of Tigress [142] (result: equiv. range equals 89.0 - 94.0%).

Conclusion. *State-of-the-art protections are not effective against black-box deobfuscation. They prevent efficient reading of the code and tracing of data but black-box methods directly execute it.*

Table 4.11: Xyntia (Xyntia$_{\text{Opt}}$) against usual protections (B2, time-out=60s)

	∅	MBA	Opaque	Path oriented	Covert channels
Succ. Rate	95.5%	95.4%	94.68%	95.4%	95.1%
Equiv. Range	90.6 - 94.2%	90.0 - 93.8%	89.9 - 93.0%	89.5- 93.7%	89.0 - 94.0%
Mean Qual.	0.92	0.95	0.90	0.94	0.89

4.7.2 Virtualization-based obfuscation

We now use Xyntia to reverse code obfuscated with state-of-the-art virtualization. We obfuscate a program computing MBA operations with Tigress [142] and VMProtect [140] and our goal is to reverse the VM handlers.[3]. Using such a synthetic program enables to expose a wide variety of handlers.

Table 4.12: Xyntia and Syntia results over program obfuscated with Tigress [142] and VMProtect [140]

		Tigress (simple)	Tigress (hard)	VMProtect
	Binary size	40KB	251KB	615KB
	# handlers	13	17	114
	# instructions per handlers	16	54	43
Xyntia	Completely retrieved	12/13	16/17	0/114
	Partially retrieved	13/13	17/17	76/114
Syntia	Completely retrieved	0/13	0/17	0/114
	Partially retrieved	13/13	17/17	76/114

[3]Note that, as Syntia, Xyntia does not consider memory operations

Tigress [142] is a source-to-source obfuscator. Our obfuscated program contains 13 handlers. Since at assembly level each handler ends with an indirect jump to the next handler to execute, we were able to extract the positions of handlers using execution traces. We then used the scripts from [67] to sample each handler. Xyntia synthesizes 12/13 handlers in less that 7 s each. We can classify them in different categories: (i) arithmetic and Boolean ($+$, $-$, $\times$, $\wedge$, $\vee$, $\oplus$); (ii) stack (store and load); (iii) control-flow (goto and return); (iv) calling convention (retrieve obfuscated function arguments). These results show that Xyntia can synthesize a wide variety of handlers. Interestingly, while these handlers contain many constant values (typically, offsets for context update), Xyntia can handle them as well. In particular, it infers the calling convention related handler, synthesizing constant values up to 28 (to access the 6th argument). Thus, even if Xyntia is inherently limited on constant values (see Section 4.5.6.2) it still handles them to a limited extent. Repeating the experiment by adding *Encode Data* and *Encode Arithmetic* to *Virtualize* yields similar results. Xyntia synthesizes all 17 exposed handlers but one, confirming that Xyntia handles combinations of protections. Finally, note that Syntia fails to synthesize handlers completely (not handling constant values). Still it infers arithmetic and Boolean handlers (without context updates).

VMProtect [140] is an assembly-to-assembly obfuscator. We use the latest premium version (v3.5.0). As each VM handler ends with a `ret` or an indirect jump, we easily extracted each distinct handler from execution traces. Our traces expose 114 distinct handlers containing on average 43 instructions (Table 4.12). VMProtect VM is stack-based. To infer the semantics of each handler, we again used Blazytko's scripts [67] in "memory mode" (i.e., forbidding registers to be seen as inputs or outputs). Our experiments show that each arithmetic and Boolean handlers (`add`, `mul`, `nor`, `nand`) are replicated 11 times to fake a large number of distinct handlers. Moreover, we are also able to extract the semantics of some stack related handlers. In the end, we successfully infer the semantics of 44 arithmetic or Boolean handlers and 32 stack related handlers. Synthesis took at most 0.3 s per handler. Syntia gets equal results as Xyntia.

Conclusion. Xyntia *synthesizes most Tigress VM handlers, (including interesting constant values) and extracts the semantics of VMProtect arithmetic and Boolean handlers. This shows that black-box deobfuscation can be highly effective, making the need for efficient protections clear.*

4.8 Counter Black-box deobfuscation

We now study defense mechanisms against black-box deobfuscation.

4.8.1 General methodology

We remind that black-box methods require the reverser to locate a suitable reverse window delimiting the code of interest with its inputs and outputs. This can be done manually or automatically [67], still this is mandatory and not trivial. The defender could target this step, reusing standard obfuscation techniques.

Still there is a risk that the attacker finds the good windows. Hence we are looking for a more radical protection against black-box attacks. We suppose that the reverse windows, inputs and outputs are correctly identified, and we seek to protected a given piece of code.

Note that adding extra *fake* inputs (not influencing the result) is easily circumvented in a black-box setting by dynamically testing different values for each input and filtering inputs where no difference is observed.

Protection rationale. Even with correctly delimited windows, synthesis can still be thwarted. Recall that black-box methods rely on 2 main steps (i) I/O sampling; (ii) learning from samples, and both can be sabotaged.

- First, if the sampling phase is not performed properly, the learner could miss important behaviors of the code, returning incomplete or even misleading information;
- Second, if the expression under analysis is too complex, the learner will fail to map inputs to their outputs.

In both cases, no information is retrieved. Hence, the key to impede black-box deobfuscation is to migrate *from syntactic complexity to semantic complexity*. We propose in Sections 4.8.2 and 4.8.3 two novel protections impeding the sampling and learning phases.

4.8.2 Semantically complex handlers

Black-box approaches are sensitive to semantic complexity. As such, relying on a set of complex handlers is an effective strategy to thwart synthesis. These complex handlers can then be combined to recover standard operations. We propose a method to generate arbitrary complex handlers in terms of size and number of inputs.

Complex semantic handlers. Let S be a set of expressions and $h, e_1, ..., e_{n-1}$ be n expressions in S. Suppose that $(S, \star)$ is a group. Then h can be encoded as $h = \overset{n-1}{\underset{i=0}{\star}} h_i$, where for all i, with $0 \leq i < n$,

$$h_i = \begin{cases} h - e_1 & \text{if } i = 0 \\ e_i - e_{i+1} & \text{if } 1 \leq i < n-1 \\ e_{n-1} & \text{if } i = n-1 \end{cases}$$

Note that $-e_i$ is the inverse element of e_i in $(S, \star)$. Each h_i is then a new handler that can be combined with others to express common operations – e.g. $x + y = h_0 + h_1 + h_2$ where $h_0 = (x + y) + -((a - x^2) - (xy))$, $h_1 = (a - x^2) - xy + (-(y - (a \wedge x)) \times (y \otimes x))$ and $h_2 = (y - (a \wedge x)) \times (y \otimes x)$. Note that the choice of $(e_1, ..., e_n)$ is arbitrary. One can choose very complex expressions with as many arguments as wanted.

Experimental design. To evaluate our new encoding, we created 3 data-sets, BP1, BP2 and BP3, listed by increasing order of complexity. Each data-set contains 15 handlers which can be combined to encode the $+, -, \times, \wedge$ and $\vee$ operators. Within a data-set, all handlers have the same number of inputs. Table 4.13 reports details on each data-set. The mean overhead column is an estimation of the complexity added to the code by averaging the number of operators needed to encode a single basic operator $(+, -, \times, \vee, \wedge)$. Overheads in BP1 (21x), BP2 (39x) and even BP3 (258x) are reasonable compared to some syntactical obfuscations: encoding $x+y$ with MBA three times in Tigress yields a 800x overhead.

Table 4.13: Protected data-sets

	#exprs	min size	max size	mean size	#inputs	mean overhead
BP1	15	4	11	6.87	3	x21
BP2	15	8	21	12.87	6	x39
BP3	15	58	142	86.07	6	x258

Evaluation. Results (Fig. 4.7) show that Xyntia (with 1 h/expr.) manages well low complexity handlers (BP1: 13/15), but performance degrades quickly

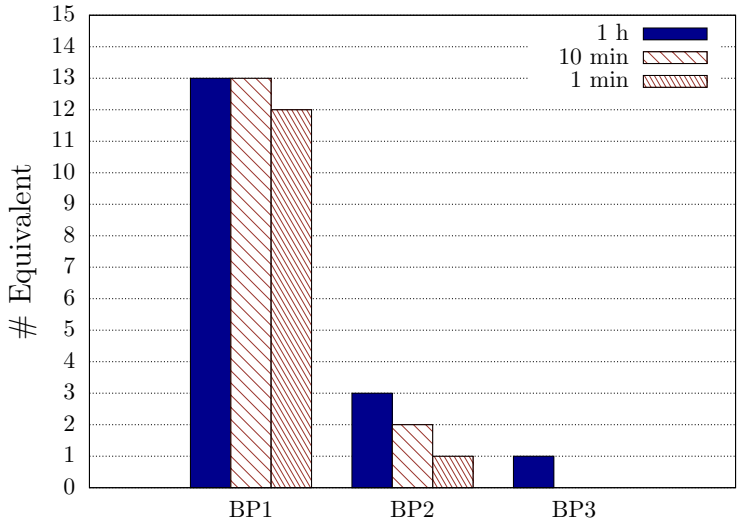


Figure 4.7: Xyntia (Xyntia$_{\text{Opt}}$) on BP1,2, 3 – varying timeouts

as complexity increases (BP2: 3/15, BP3: 1/15). Performances are similar with 1 s/expr. SYNTIA, CVC4 and STOKE-synth find none with 1 h/expr., even on BP1. Actually, SYNTIA with 12 h/expr. gets only 1/15 success of BP1.

Conclusion. *Semantically complex handlers are efficient against black-box deobfuscation. While high complexity handlers come with a cost similar to strong MBA encodings, medium complexity handlers offer a strong protection at a reasonable cost.*

Discussion. Our protection can be bypassed if the attacker focuses on the good combinations of handlers, rather than on the handlers themselves. To prevent it, complex handlers can be duplicated (as in VMProtect, see Section 4.7.2) to make patterns recognition more challenging.

4.8.3 Merged handlers

We now study another protection, based on conditional expressions and the merging of existing handlers. While block merging is known for a long time against human reversers, we show that it is extremely efficient against black-box attacks. Note that while we write our merged handlers with explicit if-then-else operators (ITE) for simplicity, these conditions are not necessarily implemented with conditional branching (cf. Fig. 4.8) Hence, we consider that the attacker sees merged handlers as a unique code fragment.

```
// if (c == cst) then h1(a,b,c) else h2(a,b,c);
int32_t res = c - cst;
res = (-((res ^ (res >> 31)) - (res >> 31)) >> 31) & 1;
return h1(a, b, c)*(1 - res) + res*h2(a, b, c);
```

Figure 4.8: Example of a branch-less condition

data-sets. We introduce 5 data-sets[4] composed of 20 expressions. Expressions in data-set 1 are built with 1 *if-then-else* (ITE) exposing 2 basic handlers (among $+, -, \times, \wedge, \vee, \oplus$); expressions in data-set 2 are built with 2 nested ITEs exposing 3 basic handlers, etc. Conditions are equality checks against consecutive constant values ($0, 1, 2$, etc.). For example, data-set 2 contains the expression:

$$ITE(z = 0, x + y, ITE(z = 1, x - y, x \times y)) \tag{4.2}$$

Scenarios. Adding conditionals brings extra challenges (i) the grammar must be expressive enough to handle conditions; (ii) the sampling phase must be efficient enough to cover all possible behaviors. Thus, we consider different scenarios:

[4]Available at : https://github.com/binsec/xyntia

Utopian The synthesizer learns expressions over the MBA set of operators, extended with an $ITE(\star = 0, \star, \star)$ operator (MBA+ITE operator set). Moreover, the sampling is done so that all branches are traversed the same number of time. This situation, favoring the attacker, will show that merged handlers are always efficient;

MBA + *ITE* This situation is more realistic: the attacker does not know at first glance how to sample. However, its grammar fits perfectly the expressions to reverse;

MBA + *Shifts* Here XYNTIA does not sample inputs uniformly over the different behaviors, does not consider ITE operators, but allows shifts to represent branch-less conditions;

Default. This is the default version of the synthesizer.

In all these scenarios, appropriate constant values are added to the grammar. For example, to synthesize Eq. (4.2), `0` and `1` are added.

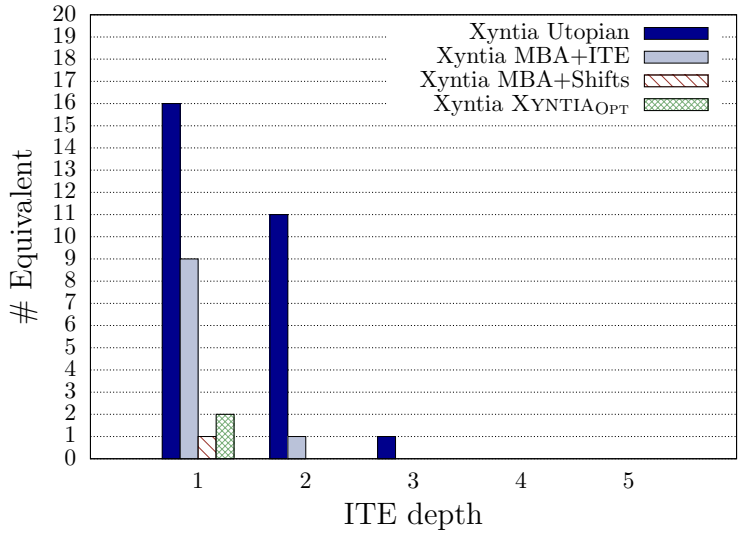


Figure 4.9: Merged handlers: XYNTIA (timeout=60s)

Evaluation. Fig. 4.9 presents XYNTIA results on the 5 data-sets. As expected, the *Utopian* scenario is where XYNTIA does best. Still, it cannot cope with more than 3 nested ITEs. For realistic scenarios, XYNTIA suffers even more. Results for SYNTIA, CVC4 and STOKE-synth confirm this result (no solution found for ≥ 2 nested ITEs). Note that overhead here is minimal, and depends only on the number of merged handlers.

Conclusion. *Merged handlers are extremely powerful against black-box synthesis. Even in the ideal sampling scenario, black-box methods cannot retrieve the semantics of expressions with more than 3 nested conditionals – while runtime overhead is minimal.*

Discussion. Symbolic methods, like symbolic execution, are unhindered by these protections, for they track the succession of handlers and know which sub parts of merged handlers are executed. To handle this, our anti-black-box protections can be combined with (lightweight) anti-symbolic protections (e.g. [146, 32]).

4.9 Related Work

Black-box deobfuscation. Blazytko et al.'s work [67] has already been thoroughly discussed. We complete their experimental evaluation, generalize and improve their approach: Xyntia with 1 s/expr. finds twice more expressions than Syntia with 600 s/expr, some of which Syntia cannot find within 12h.

White- and greybox deobfuscation. Several recent works leverage *white-box* symbolic methods for deobfuscation ("symbolic deobfuscation") [133, 134, 135, 136, 30, 137]. Unfortunately, they are sensitive to code complexity as discussed in Section 4.7, and efficient countermeasures are now available [32, 33, 138, 77] – while Xyntia is immune to them (Section 4.7.1). David et al. [73] recently proposed QSynth, a *greybox* deobfuscation method combining I/O relationship caching (black-box) and incremental reasoning along the target expression (white-box). Yet, QSynth is sensitive to massive syntactic obfuscation where Xyntia is not (cf. Section 4.6). Furthermore, QSynth works on a simple grammar. It is unclear whether its caching technique would scale to larger grammars like those of Xyntia and Syntia.

Program synthesis. Program synthesis aims at finding a function from a specification which can be given either formally, in natural language or *as I/O relations* – the case we are interested in here. There exist three main families of program synthesis methods [132]: enumerative, constraint solving and stochastic. Enumerative search does enumerate all programs starting from the simpler one, pruning snippets incoherent with the specification and returning the first code meeting the specification. We compare, in this paper, to one of such method – CVC4 [91], winner of the SyGus '19 syntax-guided synthesis competition [152] – and showed that our approach is more appropriate to deobfuscation. Constraint solving methods [94] on the other hand encode the skeleton of the target program as a first-order satisfiability problem and use an off-the-shelf SMT solver to infer an implementation meeting specification. However, it is less efficient than enumerative and stochastic methods [153]. Finally, stochastic methods [10] traverse the search space randomly in the hope of finding a program consistent with a specification. Contrary to them, we aim at solving the deobfuscation problem in a *fully* black-box way (not relying on the obfuscated code, nor on an estimation of the result size).

4.10 Conclusion

Black-box deobfuscation is a promising recent research area. The field has been barely explored yet and the pros and cons of such methods are still unclear. This article deepens the state of search-based black-box deobfuscation in three different directions. First, we define a novel generic framework for search-based black-box deobfuscation (encompassing prior works such as Syntia), we identify that the search space underlying code deobfuscation is too unstable for simulation-based methods, and advocate the use of S-metaheuristics. Second, we take advantage of our framework to carefully design Xyntia, a new search-based black-box deobfuscator. Xyntia significantly outperforms Syntia in terms of success rate, while keeping its good properties – especially, Xyntia is completely immune to the most recent anti-analysis code obfuscation methods. Finally, we propose the two first protections tailored against search-based black-box deobfuscation, completely preventing Xyntia and Syntia attacks for reasonable cost. We hope that these results will help better understand search-based deobfuscation, and lead to further progress in the field.

Part III

Conclusion and Future Work

Chapter 5

Conclusion and Future Work

Software keeps becoming bigger and more complex, making crucial tasks like code understanding, testing, and verification more and more difficult. To help users perform such tasks, automated program analysis is now required. Especially, *white-box* methods, deducing code properties from the source code syntax, are very powerful. They are notably used in large companies like Facebook, Microsoft, Amazon and Airbus. However, they also show some limitations. First, they need the source code and so, cannot be used on proprietary software where only the binary is available. Second, the complexity of the code and of the manipulated data structures impact their efficiency. Third, they are very sensitive to syntactic code complexity, which can be introduced by optimization and obfuscation passes.

In this thesis, we studied how *black-box* methods, based on artificial intelligence – from optimization to symbolic machine learning – can be used for program analysis.

5.1 Summary of our Contributions

Black-box analysis methods rely on code executions to *infer* useful code properties. They do not need the software under analysis source code and are not impacted by syntactic code complexity. Thus, they are highly suitable to help understand, test or verify complex code even if the source code is not available (proprietary software, malware, legit code).

This thesis investigates how black-box methods can benefit two hot topics from program analysis: (i) contract inference, which can help for reverse engineering, verifying or refactoring code; (ii) deobfuscation to help malicious code analysis and assess protections efficiency. Especially, we argued for the use of distinct methods to better adapt to each scenario.

- First, we propose PreCA, the first black-box function contract inference method based on constraint acquisition, a learning framework from constraint programming. It does not need the source code of the function

under analysis but only its binary version. PreCA provides contributions for both artificial intelligence and program analysis. From the artificial intelligence side, we propose the first application of constraint acquisition for program analysis. Furthermore, we extend constraint acquisition and show how to specialize it for the important problem of program analysis. We notably replace the human user with an automated oracle, solving the main constraint acquisition limitation. From the program analysis side, we propose the first black-box method with clear correctness guarantees. Being active, PreCA generates itself the queries. As such, users do not have to give a representative set of test-cases themselves, which is very hard especially if the source code is not available. We believe that it will open new research directions for both the artificial intelligence and program analysis communities;

- Second, we propose Xyntia, a new black-box deobfuscation algorithm. Xyntia corrects the flaws of the previous state-of-the-art method, seeing the deobfuscation problem as an optimization one. Especially, we advocate for the use of S-metaheuristics in place of Monte-Carlo-Tree-Search, which shows important limitations in this context. Xyntia is significantly faster and more robust to semantic complexity than the previous black-box approach. Moreover, it outperforms tested white-box methods based on rewriting rules and grey-box ones, which combine white- and black-box views. Finally, we provide the two first anti-black-box deobfuscation protections. They enable to efficiently impede black-box methods by increasing semantic complexity instead of syntactic one.

Essentially, our work shows that *black-box* analysis can efficiently infer useful code properties over complex, possibly obfuscated, code. Moreover, we show that the choice of the inference algorithm enables to find a balance between efficiency and correctness. It highlights the importance of choosing a design appropriate to the usage scenario. We believe it could open the way for new research directions in program analysis and artificial intelligence communities. In the former community, new artificial intelligence algorithms could be adapted to improve analyzers. In the later community, artificial intelligence could benefit from the program analysis application scenario, which offers new requirements, hence new algorithm trade-offs.

5.2 Perspectives

We now present some perspectives to extend the work presented in this thesis. We start with possible direct improvements of PreCA and Xyntia. Then, we propose more general and long-term research directions for black-box code analysis.

Black-box precondition inference. In Chapter 3, we proposed PreCA to infer preconditions in a black-box manner. We believe that the following improvements could be beneficial to PreCA both for speed and expressiveness:

- *Extend the bias:* PreCA takes as input a finite set of constraints, called the bias and noted B, from which it tries to express the target concept. The more expressive B is, the more confidence we can have in the result. We think that the bias could be extended in the following three directions. First, new constraints over memory access rights (is memory readable? writable? executable?) could be added. This could be very useful for reverse engineering to know for example, that one string is read but never written while another is only written. Second, it would be beneficial to design methods to integrate useful constant values in arithmetic constraints (e.g., $x \leq 100$). This could be achieved naively in the grey-box scenario, which combines black- and white-box analysis. By parsing the code, grey-box methods could extract all constant values. Still, more subtle approaches could be also considered relying on symbolic execution for example. Finally, higher-level constraints like shape [154] or complex data structures (lists, trees, graphs) could be added to apply the method to higher-level languages;

- *Handle disjunctions:* As seen in Chapter 3, preconditions can be highly disjunctive. For now, PreCA uses a simple and ad-hoc heuristic to decide the maximum size of Horn clauses considered in the bias. While it works well in our context, a more well-funded method would be beneficial. On the other hand, in the grey-box scenario, other (possibly more precise) heuristics could be used. Indeed, observe that disjunctions often arise due to conditionals and loops. Thus, one could simply parse the code or leverage symbolic methods to deduce the maximum size of Horn clauses. Another interesting research line would be to devise a more adaptive PreCA where the bias could be extended with new needed disjunctions on the fly. It could enable (in the grey-box scenario) to extract information from execution traces and add new disjunctions if, for example, a loop has been traversed or not.

Black-box deobfuscation. In Chapter 4 we proposed Xyntia to deobfuscate highly protected code blocks. We believe that there is pace for improvements:

- *Extend the grammar:* For now, Xyntia only considers expressions with bit-wise and arithmetic operators over bit-vector inputs. It would be beneficial to extend the grammar for example to synthesize loop programs, conditionals and diversify the input types (e.g., strings, arrays). This would make black-box methods more general and usable for reverse

engineering. Moreover, the grammar could be extended or modified during synthesis depending on the I/O examples results. For example, if running the code over inputs $(x = 0, y = 1)$ does not raise a *division by zero* error then expressions $expr \div x$ and $expr \div (1 - y)$ cannot be a solution. Thus we could prune part of the search space accordingly. On the other hand, if the code raises a *division by zero* error over input $(x = 5, y = 10)$, we could bias the search over expressions with a division by $x - 5$ or $y - 10$;

- *Reverse window detection:* In Chapter 4, we consider that the reverse engineer successfully found the reverse window. However, in practice, it can be hard to find, especially when anti-black-box deobfuscation is used Section 4.8. In such a scenario, it would be highly beneficial to devise methods to detect automatically reverse windows. We believe that such reverse window detection should also rely on black-box code analysis to make the approach robust against obfuscation;

- *New protections:* In Section 4.8 we propose the first protections efficient against black-box methods. However, the proposed methods are vulnerable to symbolic-based deobfuscation. As such, in order to protect efficiently, it is necessary to combine protections. Still, this can impact drastically code performance. We believe that studying new efficient protections against both white- and black-box deobfuscation is a promising research line. It could enable to extend the state-of-the-art of obfuscation with new efficient, stand-alone, and not too-costly protections. For example, implementing merged handlers through covert channels could efficiently impede both white- and black-box analysis with a limited cost.

General. Besides the proposed improvements, we believe that different research directions could be followed:

- *New applications:* We believe that a broader range of other problems could be attacked with our methods. For example, the literature proposes methods to infer postconditions or loop invariants [40, 155]. Finding black-box methods offering clear guarantees could be a great research line. Moreover, neural-based methods arose to detect malware [156, 157], infer types [78, 158], detect code similarity [80] or infer function names [79, 158, 159]. All these methods can be of great use for analysts and reverse engineers. However, they only consider what could be called "white-box features" (extracted from the code syntax). Thus, such methods are highly impacted by obfuscation and other code transformations. A promising research line could be to integrate "black-box features" to make such methods more robust;

- *Free synthesis from the user:* Black-box methods rely on programming by example (PBE). However, usual PBE considers that the number of I/O examples are given by a user and will thus stay low. In our method, the user is removed and generating examples is automated. Thus, the number of I/O examples can be high. Such an observation could lead to new approaches designed to be highly efficient when the number of I/O examples is high. For example, we found out that considering a big number of examples (100) enables to guide Xyntia more precisely. Being able to handle a large number of queries also justifies the use of PreCA-like methods;

- *Grey-box approaches:* PreCA and Xyntia only rely on observed input-output behaviors. These can be easily extracted from the binary code by running it over sampled inputs. Still, other kinds of information can be extracted using usual program analysis methods like symbolic execution or abstract interpretation. Including such white-box information to improve black-box methods could lead to more robust algorithms. For example, PreCA could be extended to ask more general queries that could be answered by static analysis. A balance should be found in order to preserve efficiency on obfuscated code;

- *Extend constraint acquisition:* Constraint acquisition queries a human user to infer the concept he has in mind. In some scenarios like program analysis, this human user can be replaced by an oracle answering queries automatically. In such contexts, constraint acquisition, whose current focus is to generate few queries, could then focus on speed. To do so, new query generation strategies and new kinds of queries relevant to the application scenario could be employed. Furthermore, ranked sets of constraints could be considered to infer incrementally target concepts. Learning would start with simple constraints first and then increase their complexity. This could help to prune known to be useless constraints quickly, speeding up convergence.

Bibliography

[1] Nancy G Leveson. "Software safety: Why, what, and how". In: *ACM Computing Surveys (CSUR)* 18.2 (1986), pp. 125–163.

[2] Gary McGraw. "Software security". In: *IEEE Security & Privacy* 2.2 (2004), pp. 80–83.

[3] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. "A survey of automated techniques for formal software verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.7 (2008), pp. 1165–1178.

[4] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. "Malware analysis and classification: A survey". In: *Journal of Information Security* 2014 (2014).

[5] Robin Gandhi et al. "Dimensions of cyber-attacks: Cultural, social, economic, and political". In: *IEEE Technology and Society Magazine* 30.1 (2011), pp. 28–38.

[6] Ralph Langner. "Stuxnet: Dissecting a cyberwarfare weapon". In: *IEEE Security & Privacy* 9.3 (2011), pp. 49–51.

[7] Daniel Dvorak. "NASA study on flight software complexity". In: *AIAA infotech@ aerospace conference and AIAA unmanned... unlimited conference*. 2009, p. 1882.

[8] Manuel Rigger et al. "An analysis of x86-64 inline assembly in c programs". In: *VEE*. 2018.

[9] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. "An empirical study of bugs in test code". In: *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE. 2015, pp. 101–110.

[10] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic superoptimization". In: *ACM SIGARCH Computer Architecture News* (2013).

[11] Christian Collberg, Clark Thomborson, and Douglas Low. *A taxonomy of obfuscating transformations*. 1997.

[12] Christian Collberg, Clark Thomborson, and Douglas Low. "Manufacturing cheap, resilient, and stealthy opaque constructs". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1998.

[13] Yongxin Zhou et al. "Information Hiding in Software with Mixed Boolean-Arithmetic Transforms". In: *Information Security Applications, 8th International Workshop, WISA 2007, Jeju Island, Korea, August 27-29, 2007, Revised Selected Papers*. 2007.

[14] Dominik Wermke et al. "A large scale investigation of obfuscation use in google play". In: *Proceedings of the 34th Annual Computer Security Applications Conference*. 2018, pp. 222–235.

[15] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. "Anything to hide? studying minified and obfuscated code in the web". In: *The world wide web conference*. 2019, pp. 1735–1746.

[16] Patrick Cousot, Michel Riguidel, and Arnaud Venet. *Device and process for the signature, the marking and the authentication of computer programs*. US Patent App. 11/133,380. Jan. 2006.

[17] J Karthik, PP Amritha, and M Sethumadhavan. "Video Game DRM: Analysis and Paradigm Solution". In: *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. IEEE. 2020, pp. 1–4.

[18] Patrick Cousot and Radhia Cousot. "Abstract interpretation frameworks". In: *Journal of logic and computation* 2.4 (1992), pp. 511–547.

[19] Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, 2021.

[20] Edmund M Clarke. "Model checking". In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer. 1997, pp. 54–56.

[21] Armin Biere et al. "Bounded model checking." In: *Handbook of satisfiability* 185.99 (2009), pp. 457–481.

[22] Cristian Cadar and Koushik Sen. "Symbolic execution for software testing: three decades later". In: *Communications of the ACM* 56.2 (2013), pp. 82–90.

[23] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)". In: *2010 IEEE symposium on Security and privacy*. IEEE. 2010, pp. 317–331.

[24] Roberto Baldoni et al. "A survey of symbolic execution techniques". In: *ACM Computing Surveys (CSUR)* 51.3 (2018), pp. 1–39.

[25] Robin David et al. "BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE. 2016.

[26] Thomas Ball et al. "SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft". In: *International Conference on Integrated Formal Methods*. Springer. 2004, pp. 1–20.

[27] Cristiano Calcagno and Dino Distefano. "Infer: An automatic program verifier for memory safety of C programs". In: *NASA Formal Methods Symposium*. Springer. 2011, pp. 459–465.

[28] Chris Newcombe et al. "How Amazon web services uses formal methods". In: *Communications of the ACM* 58.4 (2015), pp. 66–73.

[29] Jean Souyris et al. "Formal verification of avionics software products". In: *International symposium on formal methods*. Springer. 2009, pp. 532–546.

[30] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. "Symbolic deobfuscation: from virtualized code back to the original". In: *5th Conference on Detection of Intrusions and malware & Vulnerability Assessment (DIMVA)*. 2018.

[31] Sebastian Banescu et al. "Code obfuscation against symbolic execution attacks". In: *Annual Conference on Computer Security Applications, ACSAC 2016*. 2016.

[32] Mathilde Ollivier et al. "How to kill symbolic deobfuscation for free (or: unleashing the potential of path-oriented protections)". In: *Proceedings of the 35th Annual Computer Security Applications Conference*. 2019.

[33] Mathilde Ollivier et al. "Obfuscation: where are we in anti-DSE protections?(a first attempt)". In: *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering*. 2019.

[34] Dimosthenis C. Tsouros, Kostas Stergiou, and Christian Bessiere. "Omissions in Constraint Acquisition". In: *CP*. Ed. by Helmut Simonis. Springer, 2020.

[35] Rina Dechter, David Cohen, et al. *Constraint processing*. Morgan Kaufmann, 2003.

[36] Nicolas Beldiceanu and Helmut Simonis. "A Model Seeker: Extracting global constraint models from positive examples". In: *CP'12*.

[37] Mathias Paulin, Christian Bessiere, and Jean Sallantin. "Automatic design of robot behaviors through constraint network acquisition". In: *ICTAI*. 2008.

[38] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.

[39] Sumit Gulwani. "Automating string processing in spreadsheets using input-output examples". In: *ACM Sigplan Notices* 46.1 (2011), pp. 317–330.

[40] Michael D Ernst et al. "Dynamically discovering likely program invariants to support program evolution". In: *TSE* (2001).

[41] Patrick Baudin et al. "The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform". In: *Communications of the ACM* (Aug. 2021).

[42] David R Cok. "OpenJML: JML for Java 7 by extending OpenJDK". In: *NASA formal methods symposium*. Springer. 2011, pp. 472–479.

[43] Wolfgang Ahrendt et al. "The KeY platform for verification and analysis of Java programs". In: *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer. 2014, pp. 55–71.

[44] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3—where programs meet provers". In: *European symposium on programming*. Springer. 2013, pp. 125–128.

[45] Patrick Cousot et al. "The ASTRÉE analyzer". In: *European Symposium on Programming*. Springer. 2005, pp. 21–30.

[46] David A Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown Publishers, 1986.

[47] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[48] Valery A Nepomniaschy, Igor S Anureev, and AV Promskii. "Towards verification of C programs: Axiomatic semantics of the C-kernel language". In: *Programming and Computer Software* 29.6 (2003), pp. 338–350.

[49] Peter J Landin. "The mechanical evaluation of expressions". In: *The computer journal* (1964).

[50] Henry Gordon Rice. "Classes of recursively enumerable sets and their decision problems". In: *Transactions of the American Mathematical society* 74.2 (1953), pp. 358–366.

[51] Patrick Baudin et al. "The dogged pursuit of bug-free C programs: the Frama-C software analysis platform". In: *Communications of the ACM* 64.8 (2021), pp. 56–68.

[52] Charles Antony Richard Hoare. "An axiomatic basis for computer programming". In: *CACM* (1969).

[53] Edsger W Dijkstra. "A constructive approach to the problem of program correctness". In: *BIT Numerical Mathematics* (1968).

[54] Robert W Floyd. "Assigning meanings to programs". In: *Program Verification*. Springer, 1993.

[55] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[56] Yan Shoshitaishvili et al. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *IEEE Symposium on Security and Privacy*. 2016.

[57] Peter W O'Hearn. "Incorrectness logic". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–32.

[58] Babak Yadegari et al. "A generic approach to automatic deobfuscation of executable code". In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 674–691.

[59] Sébastien Bardin, Robin David, and Jean-Yves Marion. "Backward-bounded DSE: targeting infeasibility questions on obfuscated codes". In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 633–651.

[60] David Brumley et al. "Automatically identifying trigger-based behavior in malware". In: *Botnet Detection*. Springer, 2008, pp. 65–88.

[61] Johannes Kinder. "Towards static analysis of virtualization-obfuscated binaries". In: *2012 19th Working Conference on Reverse Engineering*. IEEE. 2012, pp. 61–70.

[62] Sebastian Schrittwieser et al. "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" In: *ACM Computing Surveys (CSUR)* 49.1 (2016), pp. 1–37.

[63] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. "SMT-based bounded model checking for embedded ANSI-C software". In: *IEEE Transactions on Software Engineering* 38.4 (2011), pp. 957–974.

[64] Jingxuan He et al. "Learning to Explore Paths for Symbolic Execution". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 2526–2540.

[65] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. "Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1021–1038.

[66] Patrice Godefroid. "Random testing for security: blackbox vs. whitebox fuzzing". In: *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. 2007, pp. 1–1.

[67] Tim Blazytko et al. “Syntia: Synthesizing the Semantics of Obfuscated Code”. In: USENIX *Security*. 2017.

[68] Timon Gehr, Dimitar Dimitrov, and Martin Vechev. “Learning commutativity specifications”. In: *CAV'15*.

[69] Sriram Sankaranarayanan et al. “Dynamic inference of likely data preconditions over predicates by tree learning”. In: *ISSTA*. ACM, 2008.

[70] Saswat Padhi, Rahul Sharma, and Todd Millstein. “Data-driven precondition inference with learned features”. In: *ACM SIGPLAN Notices* (2016).

[71] Saumya Debray and Jay Patel. “Reverse engineering self-modifying code: Unpacker extraction”. In: *2010 17th Working Conference on Reverse Engineering*. IEEE. 2010, pp. 131–140.

[72] Minh Hai Nguyen et al. “A hybrid approach for control flow graph construction from binary code”. In: *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*. Vol. 2. IEEE. 2013, pp. 159–164.

[73] Robin David, Luigi Coniglio, and Mariano Ceccato. “QSynth-A Program Synthesis based Approach for Binary Code Deobfuscation”. In: *BAR 2020 Workshop*. 2020.

[74] *American Fuzzy Lop (AFL)*. `https://lcamtuf.coredump.cx/afl/`.

[75] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. “Fuzzing: Challenges and Reflections.” In: *IEEE Softw.* 38.3 (2021), pp. 79–86.

[76] Moritz Schloegel et al. “Loki: Hardening code obfuscation against automated attacks”. In: *arXiv preprint arXiv:2106.08913* (2021).

[77] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. 2009.

[78] Daniel Lehmann and Michael Pradel. “Finding the dwarf: recovering precise types from WebAssembly binaries”. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022, pp. 410–425.

[79] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. “Probabilistic naming of functions in stripped binaries”. In: *Annual Computer Security Applications Conference*. 2020, pp. 373–385.

[80] Tristan Benoit, Jean-Yves Marion, and Sébastien Bardin. “Binary level toolchain provenance identification with graph neural networks”. In: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2021, pp. 131–141.

[81] Boaz Barak et al. “On the (im) possibility of obfuscating programs”. In: *Journal of the ACM (JACM)* (2012).

[82] Pierre Graux et al. "Abusing Android Runtime for Application Obfuscation". In: *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE. 2020, pp. 616–624.

[83] Jon Stephens et al. "Probabilistic Obfuscation Through Covert Channels". In: *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018*. 2018.

[84] Zohar Manna and Richard Waldinger. "A deductive approach to program synthesis". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2.1 (1980), pp. 90–121.

[85] Viktor Kuncak et al. "Complete functional synthesis". In: *ACM Sigplan Notices* 45.6 (2010), pp. 316–329.

[86] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. "Compositional program synthesis from natural language and examples". In: *Twenty-Fourth International Joint Conference on Artificial Intelligence*. 2015.

[87] Navid Yaghmazadeh et al. "SQLizer: query synthesis from natural language". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), pp. 1–26.

[88] Aditya Desai et al. "Program synthesis using natural language". In: *Proceedings of the 38th International Conference on Software Engineering*. 2016, pp. 345–356.

[89] Daniel Conrad Halbert. "Programming by example". PhD thesis. University of California, Berkeley, 1984.

[90] Rajeev Alur et al. *Syntax-guided synthesis*. IEEE, 2013.

[91] Clark Barrett et al. "CVC4". In: *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Springer, 2011. URL: `http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf`.

[92] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. "Scaling enumerative program synthesis via divide and conquer". In: *International conference on tools and algorithms for the construction and analysis of systems*. Springer. 2017, pp. 319–336.

[93] Andrew Reynolds et al. "Counterexample-guided quantifier instantiation for synthesis in SMT". In: *International Conference on Computer Aided Verification*. Springer. 2015, pp. 198–216.

[94] Susmit Jha et al. "Oracle-guided component-based program synthesis". In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. IEEE. 2010.

[95] Cormac Flanagan and K Rustan M Leino. "Houdini, an annotation assistant for ESC/Java". In: *International Symposium of Formal Methods Europe*. Springer. 2001, pp. 500–517.

[96] Angello Astorga et al. "Learning stateful preconditions modulo a test generator". In: *PLDI'19*.

[97] Pranav Garg et al. "ICE: A robust framework for learning invariants". In: *International Conference on Computer Aided Verification*. Springer. 2014, pp. 69–87.

[98] P Ezudheen et al. "Horn-ICE learning for synthesizing invariants and contracts". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–25.

[99] Rahul Sharma et al. "A data driven approach for algebraic loop invariants". In: *European Symposium on Programming*. Springer. 2013, pp. 574–592.

[100] Mohamed Nassim Seghir and Daniel Kroening. "Counterexample-guided precondition inference". In: *ESOP*. Springer. 2013.

[101] Francesca Rossi, Peter Van Beek, and Toby Walsh. "Handbook of Constraint Programming (Foundations of Artificial Intelligence)". In: (2006).

[102] Daniel Kroening and Ofer Strichman. *Decision procedures*. Springer, 2016.

[103] Armin Biere et al. "Conflict-driven clause learning sat solvers". In: *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications* (2009), pp. 131–153.

[104] Harald Ganzinger et al. "DPLL (T): Fast decision procedures". In: *International Conference on Computer Aided Verification*. Springer. 2004, pp. 175–188.

[105] Nicky Williams et al. "Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis". In: *European Dependable Computing Conference*. Springer. 2005, pp. 281–292.

[106] Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. "Not All Bugs Are Created Equal, But Robust Reachability Can Tell the Difference". In: *International Conference on Computer Aided Verification*. Springer. 2021, pp. 669–693.

[107] Gilles Barthe, Pedro R D'argenio, and Tamara Rezk. "Secure information flow by self-composition". In: *Mathematical Structures in Computer Science* 21.6 (2011), pp. 1207–1252.

[108] Florent Kirchner et al. "Frama-C: A software analysis perspective". In: *Formal Aspects of Computing* (2015).

[109] Joseph A Goguen and José Meseguer. "Security policies and security models". In: *1982 IEEE Symposium on Security and Privacy*. IEEE. 1982, pp. 11–11.

[110] Kim Marriott, Peter J Stuckey, and Peter J Stuckey. *Programming with constraints: an introduction*. MIT press, 1998.

[111] Christian Schulte and Mats Carlsson. "Finite domain constraint programming systems". In: *Foundations of Artificial Intelligence*. Vol. 2. Elsevier, 2006, pp. 495–526.

[112] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. "Design, implementation, and evaluation of the constraint language cc (FD)". In: *The Journal of Logic Programming* 37.1-3 (1998), pp. 139–164.

[113] Marie Pelleau et al. "A Constraint Solver Based on Abstract Domains". In: *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*. Ed. by Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni. Vol. 7737. Lecture Notes in Computer Science. Springer, 2013, pp. 434–454. DOI: 10.1007/978-3-642-35873-9_26. URL: https://doi.org/10.1007/978-3-642-35873-9%5C_26.

[114] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. "Automatic test data generation using constraint solving techniques". In: *ACM SIGSOFT Software Engineering Notes* 23.2 (1998), pp. 53–62.

[115] Arnaud Gotlieb. "Euclide: A constraint-based testing framework for critical c programs". In: *2009 International Conference on Software Testing Verification and Validation*. IEEE. 2009, pp. 151–160.

[116] Bertrand Meyer. "Eiffel: A language and environment for software engineering". In: *JSS* (1988).

[117] Michael Howard. *A brief introduction to the standard annotation language (sal)*. 2006.

[118] Patrice Godefroid, Shuvendu K Lahiri, and Cindy Rubio-González. "Statically validating must summaries for incremental compositional dynamic test generation". In: *SAS*. Springer. 2011.

[119] Patrick Cousot et al. "Automatic inference of necessary preconditions". In: *VMCAI'13*. Springer.

[120] Angello Astorga et al. "PreInfer: Automatic inference of preconditions via symbolic analysis". In: *DSN*. IEEE. 2018.

[121] Lingming Zhang et al. "Feedback-Driven Dynamic Invariant Discovery". In: *ISSTA*. ACM, 2014.

[122] Christian Bessiere et al. "Constraint acquisition". In: *Artificial Intelligence* (2017).

[123] Christian Bessiere et al. “Constraint acquisition via partial queries”. In: *IJCAI*. 2013.

[124] Monica Hutchins et al. “Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria”. In: *ICSE*. IEEE. 1994.

[125] David Gries. *The science of programming*. Springer Science & Business Media, 2012.

[126] Bishoksan Kafle et al. “An iterative approach to precondition inference using constrained Horn clauses”. In: *Theory and Practice of Logic Programming* (2018).

[127] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. “Program analysis as constraint solving”. In: *PLDI*. 2008.

[128] Pranav Garg et al. “Learning invariants using decision trees and implication counterexamples”. In: *ACM Sigplan Notices* (2016).

[129] Cristiano Calcagno et al. “Compositional shape analysis by means of bi-abduction”. In: *POPL*. ACM, 2009.

[130] Arnaud Lallouet et al. “On learning constraint problems”. In: *ICTAI*. IEEE. 2010.

[131] Dimosthenis C Tsouros, Kostas Stergiou, and Christian Bessiere. “Omissions in Constraint Acquisition”. In: *CP*. Springer. 2020.

[132] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. “Program synthesis”. In: *Foundations and Trends® in Programming Languages* (2017).

[133] Sebastian Schrittwieser et al. “Protecting Software Through Obfuscation: Can It Keep Pace with Progress in Code Analysis?” In: *ACM Comput. Surv.* (2016).

[134] Sébastien Bardin, Robin David, and Jean-Yves Marion. “Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes”. In: *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 633–651. DOI: 10.1109/SP.2017.36. URL: https://doi.org/10.1109/SP.2017.36.

[135] Babak Yadegari et al. “A Generic Approach to Automatic Deobfuscation of Executable Code”. In: *Symposium on Security and Privacy, SP*. 2015.

[136] David Brumley et al. “Automatically Identifying Trigger-based Behavior in Malware”. In: *Botnet Detection: Countering the Largest Security Threat*. Springer, 2008.

[137] Johannes Kinder. “Towards Static Analysis of Virtualization-Obfuscated Binaries”. In: *19th Working Conference on Reverse Engineering, WCRE*. 2012.

[138] Yongxin Zhou et al. "Information Hiding in Software with Mixed Boolean-arithmetic Transforms". In: *Proceedings of the 8th International Conference on Information Security Applications*. WISA'07. Springer-Verlag, 2007.

[139] Cameron B Browne et al. "A survey of monte carlo tree search methods". In: *IEEE Transactions on Computational Intelligence and AI in games* (2012).

[140] VM Protect Software. *VMProtect Software Protection*. `http://vmpsoft.com`. 2020.

[141] Oreans Technologies. *Themida – Advanced Windows Software Protection System*. `http://oreans.com/themida.php`. 2020.

[142] C. Collberg et al. *The Tigress C Diversifier/Obfuscator*. URL: `http://tigress.cs.arizona.edu/` (visited on 08/29/2019).

[143] Tim Blazytko et al. "Syntia: Breaking State-of-the-Art Binary Code Obfuscation via Program Synthesis". In: *Black Hat Asia* (2018).

[144] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.

[145] Helena Ramalhinho Lourenço, Olivier C Martin, and Thomas Stützle. "Iterated local search: Framework and applications". In: *Handbook of metaheuristics*. Springer, 2019.

[146] Babak Yadegari and Saumya Debray. "Symbolic Execution of Obfuscated Code". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2015.

[147] Ninon Eyrolles, Louis Goubin, and Marion Videau. "Defeating MBA-based Obfuscation". In: *Proceedings of the 2016 ACM Workshop on Software PROtection, SPRO@CCS 2016, Vienna, Austria, October 24-28, 2016*. 2016.

[148] Nicolas Falliere, Patrick Fitzgerald, and Eric Chien. "Inside the jaws of trojan. clampi". In: *Rapport technique, Symantec Corporation* (2009).

[149] Tora. *Devirtualizing FinSpy*. URL: `%7Bhttp://linuxch.org/poc2012/Tora,%20Devirtualizing%20FinSpy.pdf%7D`.

[150] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008.

[151] Xavier Leroy et al. *The OCaml system release 4.10*. 2020. URL: `https://caml.inria.fr/pub/docs/manual-ocaml/` (visited on 02/20/2020).

[152] Rajeev Alur et al. "SyGuS-Comp 2018: Results and Analysis". In: (2019). URL: `http://arxiv.org/abs/1904.07146`.

[153] Rajeev Alur et al. “Syntax-guided synthesis”. In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013.

[154] Bor-Yuh Evan Chang and Xavier Rival. “Relational inductive shape analysis”. In: *ACM SIGPLAN Notices* 43.1 (2008), pp. 247–260.

[155] Xujie Si et al. “Learning loop invariants for program verification”. In: *Advances in Neural Information Processing Systems* 31 (2018).

[156] Ömer Aslan Aslan and Refik Samet. “A comprehensive review on malware detection approaches”. In: *IEEE Access* 8 (2020), pp. 6249–6271.

[157] Junyang Qiu et al. “A survey of android malware detection with deep neural models”. In: *ACM Computing Surveys (CSUR)* 53.6 (2020), pp. 1–36.

[158] Jingxuan He et al. “Debin: Predicting debug information in stripped binaries”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 1667–1680.

[159] Jeremy Lacomis et al. “Dire: A neural approach to decompiled identifier naming”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 628–639.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Analyse de code en boîte noire pour la rétro ingénierie via acquisition de contraintes et synthèse de code
Mots clés : Apprentissage, Acquisition de contraintes, Inférence de contrats, Compréhension de code, Synthèse de code

Résumé : Les logiciels sont de plus en plus grands et complexes. Ainsi, certaines tâches comme le test et la vérification de code, ou la compréhension de code, sont de plus en plus difficiles à réaliser pour un humain. D'où la nécessité de développer des méthodes d'analyse automatique. Celles-ci sont usuellement en boîte blanche, utilisant la syntaxe du code pour déduire ses propriétés. Elles sont très efficaces mais présentent certaines limitations : le code source est nécessaire, la taille et la complexité syntaxique du code (accentuée par des optimisations et de l'obfuscation) impactent leur efficacité. Cette thèse explore comment les méthodes en boîte noire peuvent inférer des propriétés utiles pour la rétro-ingénierie. Nous étudions, tout d'abord, l'inférence de contrat de fonction qui tente d'apprendre sur quelles entrées une fonction peut être exécutée pour obtenir les sorties souhaitées. Nous adaptons l'acquisition de contraintes, en résolvant une de ses principales limitations : la dépendance à un être humain. En ressort PreCA, la première approche totalement boîte noire offrant des garanties claires de correction. PreCA est ainsi particulièrement approprié pour l'aide au développement. Nous étudions ensuite la déobfuscation, qui vise à simplifier du code obfusqué. Nous proposons Xyntia qui synthétise, via des S-métaheuristiques, une version compréhensible de blocs de code. Xyntia est plus rapide et robuste que l'état de l'art. De plus, nous proposons les deux premières protections contre la déobfuscation en boîte noire.

Title : Black-box code analysis for reverse engineering through constraint acquisition and program synthesis
Keywords : Contract inference, Code understanding, Constraint acquisition, Machine learning, Program synthesis

Abstract : Software always becomes larger and more complex, making crucial tasks like code testing, verification, or code understanding highly difficult for humans. Hence the need for methods to reason about code automatically. These are usually white-box, and use the code syntax to deduce its properties. While they have proven very powerful, they also show limitations : they need the source code, the code size and the data structures' complexity degrade their efficiency, they are highly impacted by syntactic code complexity amplified by optimizations obfuscations. This thesis explores how black-box code analysis can infer valuable properties for reverse engineering through data-driven learning. First, we consider the function contracts inference problem, which aims to infer over which inputs a code function can be executed to get good behaviors only. We extend the constraint acquisition learning framework, notably solving one of its major flaws : the dependency on a human user. It leads to PreCA, the first black-box approach enjoying clear theoretical guarantees. It makes PreCA especially suitable for development uses. Second, we consider the deobfuscation problem, which aims to simplify obfuscated code. Our proposal, Xyntia, synthesizes code block semantics through S-metaheuristics to offer an understandable version of the code. Xyntia significantly improves the state-of-the-art in terms of robustness and speed. In addition, we propose the two first protections efficient against black-box deobfuscation.