SAroP: Symbolische Ausführung resistenter opake Prädikate mittels stochastischer Unentscheidbarkeit

Teilnehmende:	Paul Baumgartner (18 J.)
Erarbeitungsort:	Hildesheim
Projektbetreuende:	Dr. Arndt Latußeck
Fachgebiet:	Mathematik/Informatik
Wettbewerbssparte:	Jugend forscht
Bundesland:	Niedersachsen
Wettbewerbsjahr:	2026

Inhaltsverzeichnis

1 Projektüberblick

In diesem Projekt wird eine neue Klasse an opaken Prädikaten präsentiert; die Klasse stochastischer Prädikate, welche sich als resisten gegenüber symbolischen Ausführungsattacken erweist. ...

2 Einleitung

Obfuskation (lat. *obfuscare*: verdunkeln) bezeichnet jede Transformation von Programmen zur Hinderung von sog. Reverse Engineering - der Analyse von Software zum Cracken, Verstehen oder Kopieren. Obfuskation kommt zum Einsatz in der Malwareentwicklung - um vor Detektion von sog. EDRS zu schützen, in der Industrie - um vor Kopien von Softwarefunktionen sowie vor Cracking zu schützen und im Militär - um dem Feind ein Verstädnis der eigenen Waffensysteme zu behindern. Da das Programm hierbei noch die Ursprüngliche Semantik beibehält kann jede Software mit genügend Zeit, Aufwand und Geld trotz Obfuskation verstanden werden. Der Sinn von Obfuskation ist also nicht die komplette Verhinderung von "Reverse Engineering", sondern vielmehr dieses wirtschaftlich unrentabel zu machen.

Diese Arbeit fokussiert sich auf eine Art der Obfuskation, der Kontrollflussobfuskation, und der meist verbreiteten Angriffsweise, um diese zu bekämpfen.

2.1 Beitrag dieser Arbeit

3 Theoretische Grundlagen

3.1 Obfuskation

Collberg et al. [1] definieren Obfuskation wie folgt:

Definition 3.1 (Obfuskation)**.** Sei $P \xrightarrow{\mathcal{T}} P'$ eine Transformation eines *Quellprogrammes* P zu einem *Zielprogramm* P'. Eine solche Transformation ist eine Obfuskation, wenn das obfuskierte Programm P' dasselbe beobachtbare Verhalten wie P für den Endnutzer aufweist.

Per Definition sind somit Nebenwirkungen (z.B. Herunterladen von neuen Daten, Dateierzeugung etc.) erlaubt, solange sie nicht vom Nutzer erfahren werden. Die präsentierte Methode dieser Publikation nutzt diese Lockerung der Einschränkungen auf obfuskierende Transformationen stark aus, wie später ersichtlich sein wird.

Um eine Obfuskationsmethode zu evaluieren werden typischerweise die Metriken *Stärke*, *Resilienz*, *Kosten* und *Tarnung* verwendet [1].

3.2 Opake Prädikate

Die folgenden Definitionen sind aus [6] sowie vom Pionierwerk [2] modifiziert übernommen. Es wird sich auf invariante opake Prädikate beschränkt. Ferner wird angenommen, dass alle Funktionen/Prädikatenterme evaluierbar sind.

Definition 3.2 (Opake Prädikate)**.** Sei $O : \Phi \to \{0, 1\}$ eine Abbildung einer Variable $\phi \in \Phi$ zu einem Prädikat. Das Prädikat $O(\phi)$ ist opak, wenn für alle $\phi \in \Phi$ gilt, dass $O(\phi)$ denselben Wert (1 oder 0 bzw. wahr oder falsch) hat.

In anderen Worten: Das Prädikat $O(\phi)$ ist opak, wenn dessen Wert für alle möglichen Parameter *a priori* bestimmt ist (also für den Programmierer bekannt ist) aber für ein Verständnis einer weiteren Person (ein Angreifer) *a posteriori* (durch Beobachtung) zu bestimmen ist.

Diese Arbeit unterscheidet zwischen zwei Arten opaker Prädikate:

Definition 3.3. Sei $O : \Phi \longrightarrow \{0, 1\}$ ein opakes Prädikat. $O(\phi)$ ist vom Typ

1. P^T, wenn für alle $\phi \in \Phi$ gilt: $O(\phi) = 1$ bzw. $wahr$.
2. P^F, wenn für alle $\phi \in \Phi$ gilt: $O(\phi) = 0$ bzw. $falsch$.

Abbildung 1: Kontrollflussgraph einer einfachen Funktion mit opakem Prädikat. Abbildung aus der Disassembly des Spiels "Overwatch" mittels IDA entnommen.

Abbildung 2: Ausschnitt des Kontrollflussgraphen einer Funktion mit vielen opaken Prädikaten. Abbildung aus der Disassembly des Spiels "Overwatch" mittels IDA entnommen.

Opake Prädikate werden in der Softwareobfuskation eingesetzt, um ein Verständnis über den Kontrollfluss des Programms zu behindern. Damit opake Prädikate als Obfuskationsmethode [1] genutzt werden können, müssen sie wiederholt angewandt werden. Dadurch entsteht ein komplexerer Kontrollflussgraph und der Angreifer weiß folglich nicht, welche Basisblöcke zu analysieren sind. Die Stärke der opaken Prädikate ist hierbei abhängig von der Stärke ihres Terms/Ausdrucks.

Beispiel 1. *Das Prädikat* $O(\phi) = -1977224191\&1 = 1$ *aus Abb. 1, wobei "&" dem bitweisen "und" Operator entspricht, ist sehr einfach. Eine Berechnung genügt, um zu erkennen, dass das Prädikat immer wahr ist.*

Mit zunehmender Komplexität der Prädikate und zunehmender Anzahl dieser, nimmt also auch die Obfuskationsstärke (Verwirrung und Unverständnis) beim Angreifer zu.

3.3 Symbolische Ausführung

Nach dem aktuellen Stand der Forschung scheint symbolische Ausführung der vielversprechendste Ansatz für die Bekämpfung opaker Prädikate zu sein. Eine Symbolische Ausführungsmaschine besteht aus 2 Hauptkomponenten: ein zentrales symbolischen Ausführungsmodul und einem Modul zur Lösung von Einschränkungen ("constraint solver"). Es gibt zwei Arten symbolischer Ausführungsmodule: statisch und dynamisch (letzteres wird auch concolische symbolische Ausführung genannt). Concolische Ausführungsmaschinen wie BAP und Triton führen zunächst das Programm mit konkreten Werten aus und führen

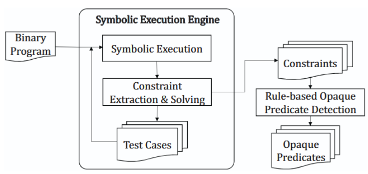

Abbildung 3: Konzeptuelles Framework zur Erkennung opaker Prädikate mit symbolischer Ausführung, Abb. aus [7] übernommen

[1]D.h, dass der wirkiche Pfad, welcher von einem opaken Prädikat verschleiert wird, nicht einfach erkannt werden kann

dann eine symbolsiche Analyse der generierten Befehlsspuren durch.
Statische symbolische Ausführungsmaschinen ”heben” zuerst die Assembly-Anweisungen des Programmes in eine abstraktere Zwischensprache an und führt dann eine symbolsiche Ausführung dessen mit statischen Analyseansätzen durch. Diese Methode wird erfolgreich in Angr verwendet.

4 Hintergrund und Motivation

Der folgende Abschnitt soll einen Überblick über existierende Arten opaker Prädikate liefern und diese in Bezug auf ihre Qualität evaulieren. Hierfür werden die Faktoren aus Abschnitt 3.1 verwendet.

	Stärke	Resilienz	Kosten	Tarnung
Obfuscator-LLVM [3]	X	trivial	sehr gering	hoch
Bi-Opake Prädikate [7]	X	mittel	hoch	hoch
Exceptions [4]	X	mittel	sehr hoch	gering
NP-schwere Probleme [**<empty citation>**]	X	gering	mittel	mittel
ungelöste Probleme [5]	X	hoch	sehr hoch	hoch

Tabelle 1: Vergleich existierende Arten opaker Prädikate.

- Obfuscator-LLVM ✓
- Bi-Opaque Prädikate ✓
- Linear Obfuscation to Combat Symbolic Execution
- Binary Code Side Effects ✓
- ”When Are Opaque Predicates Useful”
- ”Software obfuscation on a theoretical basis and its implementation,” IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences, 2003.
- (Collberg et al.) Tigress (dynamic opaque prediactes!!)

5 Ansatz

5.1 Angreifermodell

Diese Arbeit geht aufgrund der Ähnlichkeit behandelter Thematik von einem ähnlichen Angreifermodell wie [7] aus. Ein Angreifer hat direkten Zugriff auf das Programm und dessen Anweisungen. Es sei dem Angreifer hierbei nicht vorgegeben, wo und inwiefern das Programm obfuskiert ist. Der Angreifer kann das Programm nur statisch mittels symbolischer Ausführung sowie durch Untersuchung der Programmanweisungen analysieren und nicht ausführen. Solchen Situationen begegnet man z.B. bei Malware oder bei Software, welche für proprietäre unverfügbare Systeme geschrieben ist. Eine Härtung gegen weitere (dynamische) Analysemethoden des Angreifers wird in Abschnitt 9 diskutiert.

5.2 Algorithmus

Während andere präsentierte Ansätze auf schweren Problemen beruhen, welche gelöst werden müsse, um einen Pfad auszuschließen, macht dieser Algorithmus dieses Ausschließen mit herkömmlichen Methoden unmöglich. Hierfür wird mit Wahrscheinlichkeit gearbeitet. Für jedes zu generierendes opakes Prädikat wird im Programm ein Pseudozufallsvariable x generiert. Verschiedene Methoden hierfür werden in Abschnitt 6.3 gegeben. Wichtig ist, dass der Angreifer den Wert von x nicht statisch bestimmen kann. Es wird angenommen, dass x uniform verteilt ist. Diese Verteilung wird nun in eine andere (z.B. Normal-, Exponential-, Bernoulliverteilung) transformiert.

Der Pseudocode hierfür ist in Algorithmus 1 beschreiben.

Algorithm 1 Generierung stochastischer opaker Prädikate

```
Erstelle Funktion TransformiereVerteilung(x) im Modul, welche uniform verteilte Variable x ∈]0; 1] normal verteilt.

procedure FügePrädikatEin(Modul, T)
    for all Funktion in Modul do
        BasisBlock ← ZufälligerBasisBlock(Funktion)
        Entry ← ZufälligeAnweisung(BasisBlock)
        ZufaelligerWert ← ErstelleFunktionsAufruf(UniformeVerteilung, ]0; 1])

        TransformierteVariable ← ErstelleFunktionsAufruf(TransformiereVerteilung, ZufaelligerWert)
        ErstelleVergleich(TransformierteVariable)

        BB_immer_wahr ← SplitBlock(BasisBlock, Entry)        ▷ Basisblöcke für Prädikatenfälle generieren
        BB_nie_wahr ← NeuerBlock(Funktion)

        ErstelleBedingteVerzweigung("TransformierteVariable ≥ T", BB_immer_wahr, BB_nie_wahr)

        Fülle Basisblock BB_nie_wahr mit zufälligen Anweisungen.
    end for
end procedure
```

5.3 Experimente

5.3.1 Optimale Wahrscheinlichkeitsverteilung

6 Implementierung

6.1 Entscheidungen

LLVM am besten, weil viele Probleme gelöst von Experten; funktioniert auf Windows, Linux etc. (Kann zeit mit wichtigem verbringen); Sprachenunabhängig

rand() würde schlecht funktionieren, weil man SMT-solver konfigurieren könnte (auch hier hat man theoretisch gewonnen, weil sich Reverse-Engineer damit beschäftigen muss), sodass rand()=Konstante und somit alles determinierbar ist

6.2 Wahrscheinlichkeitsverteilungsgenerierung

6.3 Generierung von Pseudozufallsvariablen

6.4 Generierung von ununterscheidbarem Füllcodes

7 Evaluierung

7.1 Kriterien

Kriterien nach Collberg et al. ...

7.2 Vergleich mit existierenden Obfuskationsmethoden

8 Ergebnisdiskussion

9 Fazit und Ausblick

Literaturverzeichnis

[1] Christian Collberg, Clark Thomborson und Douglas Low. “A Taxonomy of Obfuscating Transformations”. In: *http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl?serial* (Jan. 1997).

[2] Christian Collberg, Clark Thomborson und Douglas Low. “Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs”. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL ’98*. The 25th ACM SIGPLAN-SIGACT Symposium. San Diego, California, United States: ACM Press, 1998, S. 184–196. DOI: `10.1145/268946.268962`. URL: `http://portal.acm.org/citation.cfm?doid=268946.268962` (besucht am 17. 07. 2025).

[3] Pascal Junod u. a. “Obfuscator-LLVM – Software Protection for the Masses”. In: *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO’15, Firenze, Italy, May 19th, 2015*. Hrsg. von Brecht Wyseur. IEEE, 2015, S. 3–9. DOI: `10.1109/SPRO.2015.10`.

[4] Hong Lin u. a. “Branch Obfuscation Using Binary Code Side Effects”. In: *Proceedings of the International Conference on Computer, Networks and Communication Engineering (ICCNCE 2013)*. The International Conference on Computer, Networks and Communication Engineering (ICCNCE 2013). China: Atlantis Press, 2013. DOI: `10.2991/iccnce.2013.37`. URL: `http://www.atlantis-press.com/php/paper-details.php?id=6493` (besucht am 13. 07. 2025).

[5] “Linear Obfuscation to Combat Symbolic Execution”. In: Zhi Wang u. a. *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, S. 210–226. ISBN: 978-3-642-23821-5 978-3-642-23822-2. DOI: `10.1007/978-3-642-23822-2_12`. URL: `http://link.springer.com/10.1007/978-3-642-23822-2_12` (besucht am 22. 07. 2025).

[6] Ramtine Tofighi-Shirazi u. a. *Defeating Opaque Predicates Statically through Machine Learning and Binary Analysis*. 4. Sep. 2019. DOI: `10.48550/arXiv.1909.01640`. arXiv: `1909.01640 [cs]`. URL: `http://arxiv.org/abs/1909.01640` (besucht am 23. 06. 2025). Vorveröffentlichung.

[7] Hui Xu u. a. “Manufacturing Resilient Bi-Opaque Predicates Against Symbolic Execution”. In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). Luxembourg City: IEEE, Juni 2018, S. 666–677. DOI: `10.1109/dsn.2018.00073`. URL: `https://ieeexplore.ieee.org/document/8416525/` (besucht am 17. 07. 2025).

Abbildungsverzeichnis