

PoP: Probabilistische opake Prädikate gegen symbolische Ausführung

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

Inhaltsverzeichnis

1	Projektüberblick	0
2	Einleitung	1
3	Theoretische Grundlagen	1
3.1	Obfuskation	1
3.2	Opake Prädikate	2
3.3	Symbolische Ausführung	2
4	Hintergrund und Motivation	3
5	Ansatz	4
5.1	Angreifermodell	4
5.2	Probabilistische opake Prädikate	5
5.3	Algorithmus	6
5.4	Wahrscheinlichkeitsverteilungsgenerierung	6
6	Implementierung	7
6.1	Entscheidungen	7
6.2	Generierung von Pseudozufallsvariablen	8
7	Evaluierung	8
7.1	Kriterien	8
7.2	Vergleich mit existierenden Obfuskationsmethoden	8
8	Ergebnisdiskussion	8
9	Fazit und Ausblick	8
	Literaturverzeichnis	B
	Abbildungsverzeichnis	B

1 Projektüberblick

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. *Endpoint Detection and Response* Systemen zu schützen, in der Industrie - um vor Kopien von Softwarefunktionen sowie vor Cracking zu schützen und im Militär - um dem Feind ein Verständnis 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. Von besonderem Interesse im Bereich der Obfuskation sind opake Prädikate, eine Kontrollflussobfuskation welche immer wahre bzw. falsche Verzweigungen in Programme einfügt. In dieser Arbeit wird folgender Frage nachgegangen: *Ist es Möglich, opake Prädikate zu kreieren, welche eine perfekte automatische Deobfuskation unmöglich machen?* Aufbauend auf den Gedanken von [11] wird hierfür die neue Klasse probabilistischer opaker Prädikate vorgestellt. Implementiert wird die Idee in Form eines LLVM-Passes. Eine Evaluierung anhand der Kriterien Collbergs et al. [3] zeigt, dass probabilistische opake Prädikate nicht nur resistent gegenüber symbolischer Ausführung sind, sondern auch gegenüber neuester KI-Deobfuskationsmethoden.

Diese Arbeit nimmt ein mathematisch-informatisches Grundwissen an Assembler, Kompilern sowie grundlegender Zahlentheorie an. Zudem wird die Iverson-Klammer/Prädikatabbildung $[\cdot]$ verwendet mit $[P] = 1$, falls die Aussage P wahr ist und $[P] = 0$ sonst. Vergleicht eine Aussage zwei Variablen, so wird hierfür „==“ verwendet.

3 Theoretische Grundlagen

3.1 Obfuskation

Collberg et al. [3] definieren Obfuskation wie folgt:

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

Eine Obfuskation hat immer das Ziel, die Komplexität eines Programmes so zu erhöhen, dass dessen interne Logik für einen Angreifer nur schwer verständlich ist. Für die Komplexität von Programmen gibt es mehrere Metriken [4].

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

Das Rückgängigmachen einer Obfuskation ist die *Deobfuskation*.

Um eine Obfuskationsmethode zu evaluieren werden typischerweise folgende Metriken verwendet: *Stärke* (Wie unverständlich ist die Obfuskation für einen Angreifer), *Resilienz* (Wie schwer ist eine Deobfuskation?), *Kosten* (Wie sehr erhöht die Obfuskation die Laufzeitkosten?) und *Tarnung* (Wie leicht lässt sich ein obfuskieretes Programm als solches erkennen?) [3].

3.2 Opake Prädikate

Die folgenden Definitionen sind aus [12] sowie vom Pionierwerk [4] modifiziert übernommen. Es wird sich auf hierbei auf invariante opake Prädikate beschränkt.

Definition 3.2 (Opake Prädikate). Sei $O : \Phi \rightarrow \{0, 1\}$ 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 [4].



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

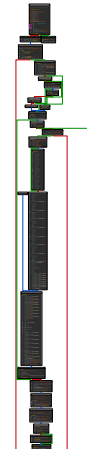


Abbildung 2: Ausschnitt des Kontrollflussgraphen einer Funktion mit vielen opaken Prädikaten. Abbildung aus der Disassembly des Spiels *Overwatch* mittels IDA entnommen.

Opake Prädikate werden in der Softwareobfuskation eingesetzt, um ein Verständnis über den Kontrollfluss des Programms zu behindern [4, 12]. Damit opake Prädikate als Obfuskationsmethode¹ 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 [4]. Mit zunehmender Komplexität der Prädikate und zunehmender Anzahl dieser, nimmt also auch die Obfuskationsstärke (Verwirrung und Unverständnis) beim Angreifer zu.

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.

Beispiel 2. Das Prädikat $O(\phi) = [(y < 10) \vee (x \cdot (x + 1) \bmod 2 \equiv 0)]$ aus [6] mit $\phi = (x, y)$ und $x, y \in \mathbb{Z}$ ist immer wahr, da $x \cdot (x + 1)$ immer gerade ist. Der Wert ist folglich von y unabhängig.

3.3 Symbolische Ausführung

Im Gegensatz zur konkreten Ausführung, welche ein Programm für spezifische Inputs ausführt, ermöglicht die symbolische Ausführung die Analyse des Programmverhaltens für ganze Klassen an Inputs [1]. Die

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

Notwendigkeit symbolischer Ausführung ergibt sich schon am Beispiel einer einer Funktion mit zwei 64-Bit Variablen. Um mit konkreter Ausführung herauszufinden, für welche Werte eine Bedingung wahr ist, müsste man hier $2^{64} \cdot 2^{64} = 2^{128}$ verschiedene Werte ausprobieren. Ein solcher Bruteforce ist selbst für die modernsten Computer unmöglich - symbolische Ausführung hingegen schon. Ein symbolischer *Ausführungsengine* besteht aus 2 Hauptkomponenten: einem *symbolischen Ausführungsmodul* und einem Constraint-Solver² zur Lösung/zum Prüfen von Bedingungen/Einschränkungen. Bei der symbolischen Ausführung wird für jeden Kontrollflussweg eine *Pfadformel* und ein *symbolischer Speicher* mitgeführt [1].

1. Die Pfadformel, eine boolesche Formel erster Ordnung, führt die Bedingungen der entlang des Pfades genommenen Verzweigungen zusammen [1].
2. Der symbolische Speicher bildet unbekannte Variablen (z.B. Parameter und alle darauf aufbauende Variablen) auf symbolische Ausdrücke ab [1].

Hierdurch können schließlich über den Constraint-Solver allgemeine Aussagen über die Erreichbarkeit bestimmter Pfade oder Variablenwerte getroffen werden [1]. Ist eine Pfadformel erfüllbar, kann der Solver zudem konkrete Eingebewerte hierfür liefern [1]. Hat das Programm aber besonders viele Verzweigungen (z.B. durch Schleifen) oder komplexe Constraints (z.B. nichtlineare Arithmetik), stoßen die Constraint-Solver an ihre laufzeittechnischen Grenzen [1]. Zur Lösung wurden verschiedene Ansätze (z.B. *concolic execution*) [1] entwickelt. Aufgrund der Fülle an Informationen und der geringen Relevanz für die in dieser Arbeit dargestellten Abwehrmethodik, wird auf ihre Darstellung verzichtet..

4 Hintergrund und Motivation

Dieser Abschnitt präsentiert den aktuellen Stand der Forschung zu opaken Prädikaten und begründet daraus diese Arbeit. Es werden aktuelle, zentrale Ansätze exemplarisch vorgestellt, um Forschungsstand und Herausforderungen zu verdeutlichen.

Existierende Literatur beschränkt sich vornehmlich auf statische Analyseansätze. Dynamische Analyseideen z.B. zur probabilistischen Untersuchung opaker Prädikate wurden veröffentlicht und experimentell untersucht, ergaben aber eine zu hohe Fehlerquote. Insbesondere reduzieren sich publizierte Ansätze auf symbolische Ausführung. Dies hat den Hintergrund, dass die symbolische Ausführung momentan eine der effektivsten automatisierten Analysemethoden bildet, welche mit wenig Aufwand und eigenem Eingriff verwendet werden kann. Andere Analysemethoden, wie z.B. *Tainting* sind zudem abhängiger von Faktoren neben den opaken Prädikaten selbst. Im Falle des *Taintings* ist die Qualität des Füllcodes wesentlich.

Trotz der Effektivität mancher existierender Methoden, bleiben viele theoretisch-formell unbegründet. Ihre Resistenz basiert auf Implementierungsschwächen³ existierender symbolischer Ausführungsengines und nicht ihren fundamentalen Grenzen (bzw. den der Constraint-Solver) [13]. Als Beispiel hierfür dienen die Bi-Opaken Prädikate [13]. Eine Befragung von Audrey Dutcher, einer der Entwicklerinnen von *Angr* [10] ergab: drei der vier in [13] dargestellten Methoden können nun von *Angr* problemlos symbolisch ausgeführt werden⁴. Die Deobfuskation weiterer Methoden wie [8] und [2] liegt also alleine in der

²Es handelt sich meist um einen SMT-Solver.

³bzw. Heuristikschwächen.

⁴(a) symbolischer RAM: Ausführbar für Arrays mit einer Länge unter 257.

```

1 int func(int symvar){
2     int j = symvar;
3     if(j == 7){
4         Foo();
5     }
6 }
7

```

(a) Ursprüngliches Programm, übernommen aus [13].

```

1 int func(int symvar){
2     int j = symvar;
3     int l1_ary[] =
4         {1,2,3,4,5,6,7};
5     int l2_ary[] = {j
6         ,1,2,3,4,5,6,7};
7     int i = l2_ary[l1_ary[j
8         %7]];
9     if(i == j)
10        Bogus();
11    if(i == 1 && j == 7)
12        Foo();
13 }
14

```

(b) Bi-Opakes Prädikat (symbolischer Arbeitsspeicher) [13].

```

1 ...
2

```

(c) TODO

```

1 ...
2

```

(d) TODO

```

1 ...
2

```

(e) TODO

Abbildung 3: Beispiele publizierter opaker Prädikate zur Verhinderung symbolischer Ausführung.

Verbesserung existierender symbolischer Ausführungseines. Eine Deobfuskation ist in gewisser Weise nur eine Frage der Zeit.

5 Ansatz

5.1 Angreifermodell

Diese Arbeit geht aufgrund der Ähnlichkeit behandelter Thematik von einem Angreifermodell aufbauend auf [13] aus. Ein Angreifer hat direkten Zugriff auf das Programm und dessen Anweisungen. Es sei dem Angreifer hierbei nicht vorgegeben, wo und inwiefern das Programm obfuskert ist. Der Angreifer kann das Programm nur statisch analysieren. Der Angreifer kann das Programm nicht im vollen Ganzen dynamisch ausführen. Solchen Situationen begegnet man z.B. bei Malware, welche gegen Virtuelle Maschinen (*virtual machines*) gehärtet ist oder bei Software, welche für proprietäre und unverfügbare Systeme geschrieben ist. Pattern matching, also das Suchen von Assembler-Anweisungsfolgen, kann hierbei zum Finden und Löschen zuvor erkannter opaker Prädikate verwendet werden.

Zudem kann der Angreifer Funktionen symbolisch ausführen. Über eine Anfrage an den Constraint-Solver kann hierbei geprüft werden, ob ein Prädikat für alle Eingabewerte wahr ist. Ist dies der Fall, so

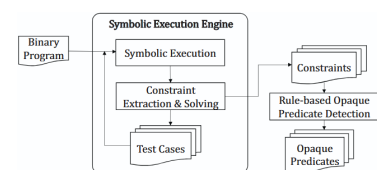


Abbildung 4: Konzeptuelles Framework zur Erkennung opaker Prädikate mit symbolischer Ausführung [13]

(b) Gleitkommazahlen: Ausführbar, wenn keine x86 long double Datentypen verwendet werden.

(c) Verdeckte symbolische Kontrollflussübertragung (*covert symbolic propagation*), in [13] über Dateisystem-Operationen implementiert: Ausführbar.

(d) Threads: Noch nicht implementiert.

handelt es sich um ein opakes Prädikat, welches gelöscht werden kann.

Eine Härtung gegen weitere (dynamische) Analysemethoden des Angreifers wird in Abschnitt 9 diskutiert.

5.2 Probabilistische opake Prädikate

Symbolische Ausführung genügt für die Untersuchung deterministischer Algorithmen und entscheidbarer Probleme. Will man aber Aussagen über einen probabilistischen Algorithmus treffen, so ist dies ohne erhebliche manuelle Eingriffe eines Nutzers in Form von extra Annahmen und Beschränkungen (*Constraints*) unmöglich. Jede zufällige Variable eines jeden Schleifenaufwurfes muss vom Constraint Solver als symbolische Variable betrachtet und somit für alle möglichen Werte überprüft werden. Bei einem Monte-Carlo-Algorithmus bedeutet dies, dass auch unwahrscheinliche Variablenwerte, welche zu falschen Ergebnissen führen, überprüft werden. Der Wahrheitsgehalt wird somit zwar formell logisch korrekt bewiesen - praktisch allerdings nicht.

Beispiel 3. Man betrachte einen Algorithmus, welcher mit einer Wahrscheinlichkeit von 99,9999999999% den Wert 1 zurückgibt und mit einer Wahrscheinlichkeit von 0,0000000001 den Wert 0 zurückgibt.

Algorithm 1 Beispiel eines probabilistischen Algorithmus

```

1: procedure Foo()
2:    $x \leftarrow \text{UNIFORMRAND}(0, 1)$ 
3:   if  $x \leq 0,999999999999$  then
4:     return 1
5:   end if
6:   return 0
7: end procedure

```

Nutzt man einen symbolischen Ausführungseengine, um zu prüfen, ob der vorliegende Algorithmus den Wert 1 wiedergibt, so würde dieser behaupten, dass dies falsch sei.

Dies bildet die Grundidee probabilistischer opaker Prädikate. Anstatt Prädikate zu bilden, welche für alle Werte ihrer Parameter *wahr* bzw. *falsch* sind, werden Prädikate erzeugt, deren gewünschter Wert so wahrscheinlich ist, dass das Gegenteil praktisch nie auftritt.

Definition 5.1 (Probabilistische opake Prädikate). Sei $O : \Phi \rightarrow \{0, 1\}$ eine Abbildung einer Variable $\phi \in \Phi$ zu einem Prädikat. Das Prädikat $O(\phi)$ ist probabilistische opak, wenn der Fall $A \in \{0, 1\}$ mit einer so hohen Wahrscheinlichkeit eintritt, dass der Fall \bar{A} vernachlässigbar ist.

Definition 5.2. Sei X eine Zufallsvariable mit beliebiger Wahrscheinlichkeitsverteilung $P(X; \theta)$, wobei θ die Parameter der Verteilung darstellen. Das probabilistische opake Prädikat $O(\phi)$ wird wie folgt konstruiert:

$$O(\phi) = [f(X) \bowtie c], \quad (1)$$

wobei:

1. f eine Transformation durch arithmetische und bitweise Operationen ist (z. B. $f(X) = m \cdot (X \oplus k) + b$), mit Konstanten $(m, k, b) \in \mathbb{R}$,
2. \bowtie ein Zahlenvergleichsoperator ist ($=, >, <, \geq, \leq$) und

3. c eine festgelegte Konstante ist.

Definition 5.3. \mathcal{P}_{prob} ist die Klasse aller probabilistische opaker Prädikate.

Beispiel 4. Das wahrscheinlich einfachste probabilistische opake Prädikat ist $O(\phi) = [UNIFORM(0, 1) \leq 1 - 10^{-2}]$. Die Wahrscheinlichkeit, dass dieses Prädikat wahr ist, beträgt $1 - 10^{-2} = 0,99\%$.

Beispiel 5. $O(\phi) = [POISSON(5) \geq 15]$. Die Wahrscheinlichkeit, dass dieses Prädikat unwahr ist beträgt $\sum_{k=15}^{\infty} \frac{5^k e^{-5}}{k!} = 1 - \sum_{k=0}^{14} \frac{5^k e^{-5}}{k!} \approx 0,023\%$

Um zu garantieren, dass das Programm, in welchem das probabilistische opake Prädikat eingefügt wurde, weiterhin funktioniert, kann für den ungewünschten Gegenfall praktische eine Wahrscheinlichkeit eingesetzt werden, welche unter der eines Hardwarefehlers liegt. Auch gewöhnliche Programme bzw. Computer können spontan versagen. Durch das Nutzen so geringer Wahrscheinlichkeiten ...

Die geringe Wahrscheinlichkeit, dass die Prädikate in \mathcal{P}_{prob} sich nicht wie gewünscht verhalten, gewährleistet ihnen theoretische Resistenz gegenüber symbolischer Ausführung. Ohne Heuristiken ist es theoretisch unmöglich, zwischen einem „normalen“ Prädikat, welches besonders häufig einen Wert annimmt, und einem probabilistische opaken zu unterscheiden.

5.3 Algorithmus

Für jedes zu generierende opakes Prädikat wird im Programm ein Pseudozufallsvariable X generiert. Verschiedene Methoden hierfür werden in Abschnitt 6.2 gegeben. Wichtig ist, dass der Angreifer den Wert von X nicht statisch bestimmen kann. Es wird angenommen, dass X gleichverteilt ist. Dies stellt ein Problem dar, da sich aus den probabilistischen Prädikaten mit dieser Zufallsvariable die Wahrscheinlichkeitswerte bereits erkennen lassen (vgl. Beispiel 4). Hierfür wird über die sog. Inversionsmethode X in eine andere, z.B. normal-, exponential- oder bernoulliverteilte, Variable transformiert. Das Vorgehen hierfür wird in Abschnitt 5.4 genauer vorgestellt. Mit dieser transformierte Zufallsvariable können nun wahrscheinliche bzw. unwahrscheinliche probabilistische Prädikate erstellt werden.

Der Pseudocode für die Generierung solcher probabilistischer opaker Prädikate ist in Algorithmus 2 beschreiben.

Algorithm 2 Generierung probabilistischer opaker Prädikate

```
1: procedure ...(...)
2: end procedure
```

5.4 Wahrscheinlichkeitsverteilungsgenerierung

Um den Ansatz gegen Pattern Matching resistent zu machen, soll dieser möglichst generalisiert werden. Anstatt sich auf eine Wahrscheinlichkeitsdichtefunktion bzw. Umkehrfunktion der kumulativen Verteilungsfunktion zu beschränken, soll für jedes probabilistisches opakes Prädikat eine neue Wahrscheinlichkeitsverteilung verwendet werden. Mehrere Methoden kommen hierfür infrage:

Generierung über Verteilungsfamilie Eine Möglichkeit ist die Nutzung einer Wahrscheinlichkeitsverteilung, deren Wahrscheinlichkeits-(dichte-)funktion über einen oder mehrere Parameter bestimmt wird.

Ein Beispiel hierfür ist die Gammaverteilung mit Skalenparameter $\alpha > 0$, Formparameter $r > 0$ und folgender Dichtefunktion [5]:

$$\gamma_{\alpha,r}(x) = \frac{\alpha^r}{\Gamma(r)} x^{r-1} e^{-\alpha x}, x > 0. \quad (2)$$

Diese hat den Vorteil, dass sich verschiedene andere Verteilungen (z.B. Chi-Quadrat-, Erlang und Exponentialverteilung) aus ihr ergeben. Ein *Pattern-Matching* Angriff wird hierdurch erschwert, da man nach Termen der sehr allgemeinen Form $ax^n b^x$ suchen müsste. **Der Nachteil hiervon ist, dass ...**

Generierung über zufälliges Polynom Eine Alternative ist die eigenständige Generierung einer zufälligen Funktion F , welche die Eigenschaften einer kumulativen Verteilungsfunktion (vgl. ??) erfüllt. Die Nutzung eines einfachen Polynoms genügt hier nicht, da diese durch ihre häufigen Oszillationen kein notwendiges streng monotonen Steigen garantieren (vgl. ??, Bedingung 1). Dies wird über Bernsteinpolynome der folgenden Form gelöst:

$$B_n(x) = \sum_{k=0}^n c_k \binom{n}{k} x^k (1-x)^{n-k}, \text{ mit } c_0 \leq c_1 \leq \dots \leq c_n \text{ und } n \in \mathbb{R}. \quad (3)$$

Um alle Eigenschaften einer kumulativen Verteilungsfunktion zu erfüllen, muss $B_n(x)$ zudem durch die Punkte $(0|0)$ und $(0|1)$ verlaufen. Hierfür genügt es, $c_0 = 0$ und $c_n = 1$ festzulegen, da: **TODO: Begründung/Beweis** Für eine weitere Verallgemeinerung lassen sich des Weiteren der x-Achsenstreckfaktor a sowie der x-Achsenverschiebungssummand k in $B_n(a \cdot (x - k))$ einfügen.

Das Vorgehen für diese Methode ist somit Folgendes:

1. Wähle zwei zufällige rationale Zahlen $(a, k) \in \mathbb{R}$.
2. Bestimme das Intervall $[x_1, x_2]$ mit $x_1 = k$ und $x_2 = k + \frac{1}{a}$.
3. Teile $[x_1; x_2]$ in n Teile ein.
4. Sei $c_0 = 0$ und $c_n = 1$. Für Intervallteil $i = 1$ bis $n - 1$:
Wähle eine zufällige rationale Zahl c . Berechne $c_i = c_{i-1} + c$.
5. Generiere die Funktion $B_n(x) = \sum_{k=0}^n c_k \binom{n}{k} x^k (1-x)^{n-k}$ mit gegebenen Parametern.

6 Implementierung

6.1 Entscheidungen

Zur Implementierung wurden drei Ansätze erwogen: die Entwicklung eines Bin2Bin-Obfuskators⁵, die Implementierung in Form eines LLVM-Passes [7, 9]⁶ sowie die Quellcodemanipulation⁷. Es wurde eine Entscheidung für einen LLVM-Pass getroffen aufgrund folgender Vorteile:

1. **Abstraktion und Portabilität:** LLVM entkoppelt durch eine abstrakte Zwischensprache (*Intermediate Representation*) von Architektur-/Betriebssystemdetails. Viele *low-level* Aufgaben (z.B. *relocations*, Einfügen von Assembler-Anweisungen etc.) werden übernommen.

⁵Direkte Manipulation von Assembler-Anweisungen existierender Programme.

⁶Nutzung des LLVM-Projekts, um einen sog. *Compiler-Pass* zu schreiben.

⁷durch z.B. C-Makros und das Einfügen von inline Assembler-Ausschnitten

2. **Optimierung:** Die LLVM-Toolchain enthält etablierte Optimierungs-Pässe und profitiert fortlaufend von der Arbeit zahlreicher Beitragender. Dies ermöglicht eine nahezu optimale Kompilation obfuskiertter Programme, welche sich als nützlich und sogar erforderlich in vielen Anwendungssituationen erweist (z.B. *Embedded Systems*, IoT, Echtzeitsysteme etc.).
3. **Entwicklungsaufwand:** Bin2Bin-Obfuskatoren und umfangreiche Quellcodemanipulation erfordern viel manuellen Aufwand und sind fehlerhaftig. Ein LLVM-Pass ermöglicht den reinen Fokus auf die Obfuskationslogik.

Ein LLVM-Pass bietet in diesem Fall das beste Kompromissverhältnis der *low-level* Kontrolle sowie Laufzeiteffizienz und der *high-level* Portabilität und geringem Entwicklungsaufwand.

6.2 Generierung von Pseudozufallsvariablen

Damit der vorgestellte Ansatz funktionieren kann, bedarf er einer (Pseudo-)Zufallszahl, welche auf dem Einheitsintervall liegt und zugleich als unbestimmte symbolische Variable von symbolischen Ausführungseines betrachtet wird. Hierfür werden die Parameter der obfuskierten Funktionen verwendet. Die Idee hierhinter ist, dass um diese zu bestimmen, ... Hat die zu obfuskierte Funktion kein Parameter, so kann ein Pseudoparameter hinzugefügt werden. Um zu garantieren, dass die generierte Pseudozufallszahl im Einheitsintervall liegt, existieren mehrere Methoden. Es wurde sich für die einfachste entschieden. Hier wird der erste Parameter als 64-Bit Gleitkommazahl betrachtet und durch dessen maximalen Wert geteilt.

Betriebssystem-APIs

Parameter-Sampling

Zeit

Race-Conditions

Hardware

7 Evaluierung

7.1 Kriterien

7.2 Vergleich mit existierenden Obfuskationsmethoden

8 Ergebnisdiskussion

9 Fazit und Ausblick

Literaturverzeichnis

- [1] Roberto Baldoni u. a. *A Survey of Symbolic Execution Techniques*. Mai 2018. DOI: 10.48550/arXiv.1610.00502. eprint: 1610.00502 (cs). (Besucht am 27. 07. 2025).
- [2] Christian Collberg. *The Tigress C Obfuscator*. <https://tigress.wtf/>. 2025. (Besucht am 14. 10. 2025).
- [3] 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).
- [4] 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).
- [5] Hans-Otto Georgii. *Stochastik: Einführung in die Wahrscheinlichkeitstheorie und Statistik*. 5. Auflage. De Gruyter Studium. Berlin ; Boston: De Gruyter, 2015. 438 S. ISBN: 978-3-11-035969-5.
- [6] 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.
- [7] Chris Lattner und Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. San Jose, CA, USA, 2004, S. 75–86. DOI: 10.1109/CGO.2004.1281665.
- [8] Hong Lin u. a. “Branch Obfuscation Using Binary Code Side Effects”. In: *Proceedings of the International Conference on Computer, Networks and Communication Engineering (ICCNC 2013)*. The International Conference on Computer, Networks and Communication Engineering (ICCNC 2013). China: Atlantis Press, 2013. DOI: 10.2991/iccnc.2013.37. URL: <http://www.atlantis-press.com/php/paper-details.php?id=6493> (besucht am 13. 07. 2025).
- [9] LLVM Project. *The LLVM Compiler Infrastructure*. <https://llvm.org/>. Version 20.1.4. 2003–2025. (Besucht am 18. 08. 2025).
- [10] Yan Shoshitaishvili u. a. “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In: *IEEE Symposium on Security and Privacy*. 2016.
- [11] Jon Stephens u. a. “Probabilistic Obfuscation Through Covert Channels”. In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018 IEEE European Symposium on Security and Privacy (EuroS&P). London: IEEE, Apr. 2018, S. 243–257. ISBN: 978-1-5386-4228-3. DOI: 10.1109/EuroSP.2018.00025. URL: <https://ieeexplore.ieee.org/document/8406603/> (besucht am 14. 10. 2025).
- [12] 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.

- [13] 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

1	Kontrollflussgraph einer einfachen Funktion mit opakem Prädikat. Abbildung aus der Disassembly des Spiels <i>Overwatch</i> mittels IDA entnommen.	2
2	Ausschnitt des Kontrollflussgraphen einer Funktion mit vielen opaken Prädikaten. Abbildung aus der Disassembly des Spiels <i>Overwatch</i> mittels IDA entnommen.	2
3	Beispiele publizierter opaker Prädikate zur Verhinderung symbolischer Ausführung. . . .	4
4	Konzeptuelles Framework zur Erkennung opaker Prädikate mit symbolischer Ausführung [13]	4