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 7
5.5 Generierung von Pseudozufallsvariablen 8
5.6 Füllcode 10

6 Implementierung **11**

7 Evaluierung **11**
7.1 Vorgehen 11
7.2 Evaluierung 13
7.2.1 Kosten 13
7.2.2 Stärke 14
7.2.3 Resilienz 14
7.2.4 Tarnung 14
7.3 Vergleich mit existierenden Obfuskationsmethoden 14

8 Ergebnisdiskussion **14**

9 Fazit und Ausblick **14**

Literaturverzeichnis **C**

Abbildungsverzeichnis **C**

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 (mit aktuellen Methoden) unmöglich machen?* Aufbauend auf den Gedanken von [21] wird hierfür die neue Klasse probabilistischer opaker Prädikate vorgestellt. Implementiert wird die Idee in Form eines LLVM-Passes. Dabei wird sich auf die Erzeugung der opaken Prädikate selbst beschränkt. Die Generierung von Füllcode (sog. *Junkcode*) wird nicht behandelt. Eine Evaluierung anhand der Kriterien Collbergs et al. [7] zeigt, dass probabilistische opake Prädikate nicht nur resistent gegenüber symbolischer Ausführung sind, ohne dabei unvertretbare Kosten aufzuweisen.

Diese Arbeit nimmt ein mathematisch-informatisches Grundwissen an Assembler, Kompilern sowie grundlegender Zahlentheorie an. Zudem wird die Iverson-Klammer/Prädikatabbildung $[\cdot]$ verwendet: Unter der Voraussetung, dass die Aussage P wahr ist, gilt $[P] = 1$. Ansonsten gilt $[P] = 0$. Pseudocode, welcher Anweisungen modifiziert nimmt an, dass diese in *Single Static Asignment*-Form definiert sind. TODO: Klären, was in welchem Abschnitt gemacht wird

3 Theoretische Grundlagen

3.1 Obfuskation

Collberg et al. [7] definieren Obfuskation wie folgt:

Definition 3.1 (Obfuskation)**.** Sei $P \xrightarrow{\mathcal{T}} P'$ eine Transformation $\mathcal{T}$ 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. TODO: Barak et al. nutzen!

Obfuskation zielt darauf ab, 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 [8]. Näheres hierzu folgt in Abschnitt 7.1.

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*.

3.2 Opake Prädikate

Die folgenden Definitionen sind aus [23] sowie vom Pionierwerk [8] 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 [8].

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

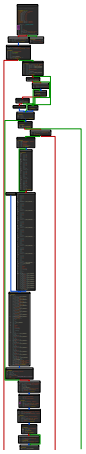


Abbildung 2: Ausschnitt des Kontrollflussgraphen einer Funktion mit vielen opaken Prädikaten. Durch die vielen opaken Prädikate wird die Funktion unübersichtlich und eine Analyse aufgrund der Unklarheit tatsächlich ausführbarer Pfade erschwert. Abbildung aus der Disassembly des Spiels *Overwatch* mittels IDA entnommen.

Opake Prädikate werden in der Softwareobfuskation eingesetzt, um ein Verständnis über den Kontrollfluss des Programms zu behindern [8, 23]. 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 [8]. Mit zunehmender Komplexität der Prädikate und zunehmender Anzahl dieser, nimmt also auch die Obfuskationsstärke (Verwirrung und Unverständnis) beim Angreifer zu (vgl. Abb. 2).

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* $O(\phi) = [(y < 10) \vee (x \cdot (x + 1) \mod 2 \equiv 0)]$ *aus [11] mit* $\phi = (x, y)$ *und* $x, y \in \mathbb{Z}$ *ist immer wahr, da* $x \cdot (x + 1)$ *immer gerade ist. Der Wert is 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

[1] D.h, dass der wirkiche 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-Solver[2] 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. Die geschieht anhand der Kriterien aus Abschnitt 7.1.

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 Implementierungsschwächen[3] existierender symbolischer Ausführungsengines und nicht ihren fundamentalen Grenzen (bzw. den der Constraint-Solver) [24]. Als Beispiel hierfür dienen die Bi-Opaken Prädikate [24]. Eine Befragung von Audrey Dutcher, einer der Entwicklerinnen von *Angr* [20] ergab: drei der vier in [24] dargestellten Methoden können nun von *Angr* problemlos symbolisch

[2] Es handelt sich meist um einen SMT-Solver.

[3] bzw. Heuristikschwächen.

```c
int func(int symvar){
    int j = symvar;
    if(j == 7){
        Foo();
    }
}
```

(a) Ursprüngliches Programm, übernommen aus [24].

```c
int func(int symvar){
    int j = symvar;
    int l1_ary[] =
    {1,2,3,4,5,6,7};
    int l2_ary[] =
      {j,1,2,3,4,5,6,7};
    int i = l2_ary[l1_ary[j%7]];
    if(i == j)
        Bogus();
    if(i == 1 && j == 7)
        Foo();
}
```

(b) Bi-Opakes Prädikat (symbolischer Arbeitsspeicher) [24].

```
...
```

(c) TODO

```
...
```

(d) TODO

```
...
```

(e) TODO

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

ausgeführt werden [4]. Die Deobfuskation weiterer Methoden wie [13], [6] und [13] liegt also alleine in der Verbesserung existierender symbolischer Ausführungsengines. Eine Deobfuskation ist in gewisser Weise nur eine Frage der Zeit.

Andere Methoden, wie [14] und [17] sind zwar formal sicher, sind aber einfach erkennbar und weisen hohe Laufzeitkosten auf.

Angesichts der Schwierigkeiten aktueller Methoden ergibt sich die Notwendigkeit effizienter, getarnter und resilienter (nur schwer automatisch deobfuskierbarer) opaker Prädikate.

5 Ansatz

5.1 Angreifermodell

Diese Arbeit geht aufgrund der Ähnlichkeit behandelter Thematik von einem *Man-at-the-End* (MATE) Angreifermodell aus, aufbauend auf [24, 25]. Ein Angreifer hat direkten Zugriff auf das Programm und dessen Anweisungen sowie volle Kontrolle über das Endsystem, auf dem sie ausgeführt werden. Es ist dem Angreifer hierbei nicht vorgegeben, wo und inwiefern das Programm obfuskiert ist. Der Angreifer kann das Programm statisch und dynamisch analysieren. Das Ziel des Angreifers ist dabei, ein Verständnis der obfuskierten Programmlogik zu gewinnen. Pattern Matching, also das Suchen von Assembler-Anweisungsfolgen, kann hierbei zum Finden und Löschen zuvor erkannter opaker Prädikate verwendet werden.

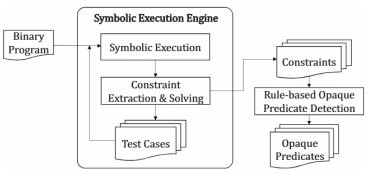


Abbildung 4: Konzeptuelles Framework zur Erkennung opaker Prädikate mit symbolischer Ausführung [24]

[4] (a) symbolischer RAM: Ausführbar für Arrays mit einer Länge unter 257.
(b) Gleitkommazahlen: Ausführbar, wenn keine x86 `long double` Datentypen verwendet werden.
(c) Verdeckte symbolische Kontrollflussübertragung (*Covert Symbolic Propagation*), in [24] über Dateisystem-Operationen implementiert: Ausführbar.
(d) Threads: Noch nicht implementiert.

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

Die Bedeutung weiterer dynamischer Analysemethoden 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 Schleifenaufrufes 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 formal logisch korrekt bewiesen - praktisch allerdings nicht.

Beispiel 3. *Man betrachte einen Algorithmus, welcher mit einer Wahrscheinlichkeit von* $99,9999\%$ *den Wert* 1 *zurückgibt und mit einer Wahrscheinlichkeit von* $0,0001$ *den Wert* 0 *zurückgibt.*

Algorithm 1 Beispiel eines probabilistischen Algorithmus

```
1: procedure Foo()
2:     X ← UniformRand(0, 1)
3:     if X ≤ 0,999999 then
4:         return 1
5:     end if
6:     return 0
7: end procedure
```

Nutzt man einen symbolischen Ausführungsengine, 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 $O_p : \Phi \rightarrow \{0,1\}$ eine Abbildung einer Variable $\phi \in \Phi$ zu einem Prädikat. Das Prädikat $O_p(\phi)$ ist probabilistische opak, wenn der Fall $A \in \{0,1\}$ mit einer so hohen Wahrscheinlichkeit eintritt, dass der Fall $\overline{A}$ vernachlässigbar ist. TODO: formeller mit ϵ wie Barak et al.

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

$$O_p(\phi) = [f(X) \bowtie c]\,, \tag{1}$$

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. *Ein einfaches probabilistisches opakes Prädikat ist* $O_p(\phi) = \left[\textsc{Uniform}(0,1) \leq 1 - 10^{-2}\right]$. *Die Wahrscheinlichkeit, dass dieses Prädikat wahr ist, beträgt* $1 - 10^{-2} = 0,99\%$.

Beispiel 5. $O_p(\phi) = [\textsc{Poisson}(5) \geq 15]$. *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 (bei adäquater Implementierung) theoretisch unmöglich, zwischen einem „normalen“ Prädikat, welches besonders häufig einen Wert annimmt, und einem probabilistischen opaken Prädikat zu unterscheiden.

Beispiel 6. *Sei p ein Prädikat, welches zu 95% der Zeit wahr ist. Ist $p \in \mathcal{P}_{prob}$ oder einfach besonders häufig wahr?*

Dies zwingt den Angreifer zu einer genaueren Analyse jedes Prädikats im Sachzusammenhang.

5.3 Algorithmus

Algorithm 2 Generierung probabilistischer opaker Prädikate

Require: D_{start}, D_{end} (Domain of inverse CDF), $P(TrueBB)$ (Probability of generated predicate having wanted value), *Precision* (determines *Threshold* precision in predicate)

procedure GenerateProbabilisticPredicates($Module, CDF, p \in [0,1], Precision \in \mathbb{N}$)
  for $Function\ F \in Module$ **do**
    if ShouldObfuscate(F) **then**
      $BB \leftarrow$ GetRandomBasicBlock(F)
      $U \leftarrow$ BB.InsertSymbolicVariable() ▷ Generate a random variable $\in [0;1]$.
      BB.InsertCallInverseCDF(U)
      $Threshold \leftarrow \frac{D_{end} - D_{start}}{2}$ ▷ Compute threshold using the Newton–Raphson method.
      for $i \leftarrow 0$ **to** *Precision* **do**
        $OffsetY \leftarrow$ CDF.EvaluateAt($Threshold$) $- P(TrueBB)$
        $Slope \leftarrow$ CDF.EvaluateDerivativeAt($Threshold$) ▷ Evaluate PDF.
        $Threshold \leftarrow Threshold - \frac{OffsetY}{Slope}$
      end for
      $RealBB \leftarrow$ BB.Split($LastInstruction$) ▷ Always executed Basicblock
      $FakeBB \leftarrow$ F.CreateBB() ▷ Never executed Basicblock
      BB.InsertIf($U < Threshold, RealBB, FakeBB$)
      FakeBB.InsertJunkCode()
    end if
  end for
end procedure

Für jedes zu generierendes opakes Prädikat wird im Programm ein Pseudozufallsvariable U generiert. Verschiedene Methoden hierfür werden in Abschnitt 5.5 gegeben. Wichtig ist, dass der Angreifer den Wert von U nicht statisch bestimmen kann. Es wird angenommen, dass U gleichverteilt ist. Dies stellt ein Problem dar, da sich aus den probabilistischen opaken Prädikaten ohne Transformationen ($F(X) = X$) mit dieser Zufallsvariable die Wahrscheinlichkeitswerte direkt herauslesen lassen (vgl. Beispiel 4). Hierfür wird über die sog. Inversionsmethode U in eine andere z.B. normal-, exponential- oder bernoulliverteilte Zufallsvariable transformiert. Das Vorgehen hierfür wird in Abschnitt 5.4 genauer vorgestellt. Mit dieser transformierten Zufallsvariable können nun wahrscheinliche bzw. unwahrscheinliche probabilistische Prädikate erstellt werden.

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

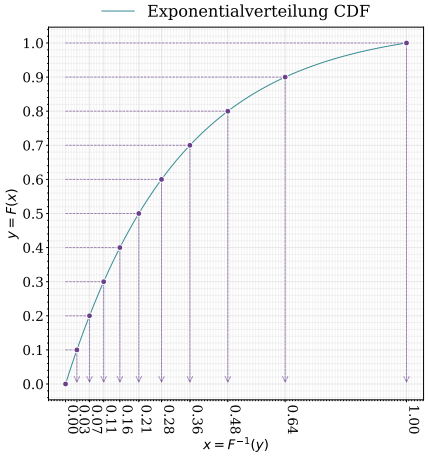


Abbildung 5: Zufällige Zahlen y_i werden von einer Gleichverteilung $Unif(0, 1)$ generiert. Jeder zufällige y-Wert wird über die inverse kumulative Verteilungsfunktion der Exponentialverteilung $F^{-1}(x)$, einem x-Wert zugeordnet. Wie bei einer Exponentialverteilung sammeln sich hierdurch die $x = F^{-1}(x)$-Werte um 0.

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 [10]:

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

Diese hat den Vorteil, dass sich verschiedene andere Verteilungen (z.B. Chi-Quadrat-, Erlang und Exponentialverteilung) aus ihr ergeben. Ein Pattern Matching Angriff müsste demnach nach Termen der sehr allgemeinen Form $ax^n b^x$ suchen[5]. Dennoch ist es nicht unmöglich: Es ist nicht davon auszugehen, dass der Term in regulären Programmen (oft/überhaupt) vorkommt.

Generierung über zufälliges Polynom Eine Alternative ist die eigenständige Generierung einer zufälligen Funktion F, welche die Eigenschaften einer kumulativen Verteilungsfunktion erfüllt. Eine kumulative Verteilungsfunktion $F : \mathbb{D} \to [0; 1]$ muss folgende 3 Eigenschaften erfüllen:

1. F ist monoton steigend.

[5]Die Faktoren werden im Vorhinein zur Kompilationszeit miteinander multipliziert

2. F ist rechtsseitig stetig.

3. $\lim_{x \to \inf \mathbb{D}+} F(x) = 0$ und $\lim_{x \to \sup \mathbb{D}-} F(x) = 1$.

Der einfachste Weg, strenge Monotonie sowie die beschriebenen Grenzwerte umzusetzen, ist über Bernsteinpolynome folgender Form. Die Umsetzung über Polynome in der Standardbasis wäre durch deren häufigen Oszillationen erschwert.

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

Um alle Eigenschaften eine 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 zitieren.

Für eine Verallgemeinerung lassen sich 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. Wähle den Definitionsbereich $\mathbb{D} = [x_1, x_2]$ 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.

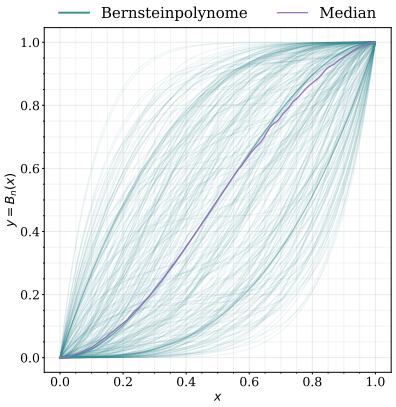


Abbildung 6: Zufällige Bernsteinpolynome fungieren als verschiedene kumulative Verteilungsfunktionen für die probabilistischen opaken Prädikate.

Das Inverse der kumulativen Verteilungsfunktion lässt sich effizient über das Newton-Raphson-Verfahren in $O(k)$ berechnen, wobei k die Anzahl an Iterationen festlegt. Aufgrund des quadratischen Konvergenzverhaltens des Verfahrens genügen für 64-Bit Gleitkommazahlen praktisch $k = 6$ Iterationen: Gemäß der IEEE 754 Norm haben 64-Bit Gleitkommazahlen eine 52-Bit Mantisse. Durch das Konvergenzverhalten verdoppelt sich die Anzahl korrekter Bits mit jeder Iteration. Ist anfänglich 1 Bit korrekt erraten, so braucht es 6 Iterationen, um alle $2^6 = 64 > 53$ Bits der Mantisse korrekt zu bestimmen. Um Sicherheitsgarantien zu liefern wurde der Algorithmus mit $k = 12$ implementiert, um bei schlechten anfänglichen Schätzungen und ungünstigen Bernsteinpolynomen dennoch garantiert das Inverse der kumulativen Verteilungsfunktion korrekt zu bestimmen. Die zusätzlichen Laufzeitkosten hierdurch sind minimal. Um eine mögliche Divergenz des Newton-Raphson-Verfahrens zu verhindern, wird der geschätzte Wert bei jeder Iteration auf den Definitionsbereich $\mathbb{D}$ der Funktion begrenzt. Der Pseudocode hierfür ist in Algorithmus 3 gegeben.

5.5 Generierung von Pseudozufallsvariablen

TODO: kürzen!!! und Pseudocode hierfür einfügen Damit der vorgestellte Ansatz funktionieren kann, bedarf er einer gleichverteilten (Pseudo-)Zufallszahl, welche auf dem Einheitsintervall liegt und zugleich

Algorithm 3 Berechnung der inversen kumulativen Verteilungsfunktion über das Newton-Raphson-Verfahren

Require: C (Bernstein coefficients), *HShift*, *VStretch* (horizontal shift/ vertical stretch), *Precision* (determines precision of Bernstein evaluation)

function SAMPLEBERNSTEININVERSE($U \in [0, 1]$)
  $Estimate \leftarrow HShift + \frac{0.5}{VStretch}$ ▷ Start guess at center of domain.
  $D_{start} \leftarrow HShift$
  $D_{end} \leftarrow HShift + \frac{1}{VStretch}$
  for $i \leftarrow 0$ **to** *Precision* **do**
    $t \leftarrow VStretch \cdot (Estimate - HShift)$ ▷ Transformed *Estimate*.
    $y \leftarrow 0.0$ ▷ Accumulator for $B(t)$.
    $y' \leftarrow 0.0$ ▷ Accumulator for $B'(t)$.
    for $k \leftarrow 0$ **to** *Degree* **do** ▷ Evaluate CDF $B(t)$
      $b_k \leftarrow \binom{n}{k} \cdot t^k \cdot (1-t)^{n-k}$
      $y \leftarrow y + C[k] \cdot b_k$
    end for
    for $k \leftarrow 0$ **to** *Degree* $- 1$ **do** ▷ Evaluate PDF.
      $b'_k \leftarrow \binom{n-1}{k} \cdot t^k \cdot (1-t)^{n-1-k}$
      $y' \leftarrow y' + C'[k] \cdot b'_k$
    end for
    $OffsetY \leftarrow y - U$
    $Slope \leftarrow y' \cdot HStretch$ ▷ Apply chain rule: $\frac{dy}{dx} = \frac{dy}{dt} \cdot \frac{dt}{dx}$.
    $Estimate \leftarrow Estimate - \frac{OffsetY}{Slope}$ ▷ Newton Step.
    $Estimate = \text{MAX}(\text{MIN}(Estimate, D_{start}), D_{end})$ ▷ Clamp *Estimate* so that Newton-Raphson doesn't diverge.
  end for
  return *Estimate*
end function

als unbestimmte symbolische Variable ohne konkreten Wert von symbolischen Ausführungsengines betrachtet wird. Infrage kommen hierfür:

1. Betriebssystem- (z.B. `rand() oder time()`) und Hardware-APIs (z.B. `rdtsc`),
2. die Zufallszahlgenerierung mittels Race Conditions [5] oder
3. die Nutzung von Funktionsparametern, deren Werte sich nicht statisch festlegen lassen (*Parameter Sampling*).

In Fällen 1-2 sind die genutzten Quellen nach erstmaliger Bestimmung leicht wiedererkennbar. Dies erleichtert für Angreifer die Suche nach probabilistischen opaken Prädikaten und mindert folglich auch ihre Tarnung.

Es wurde sich daher für Methode 3 entschieden. In der Praxis können Angreifer kaum ganze Programme symbolisch auszuführen sondern immer nur einzelne Funktionen. Je größer der Abschnitt, welcher symbolisch ausgeführt wird, desto höher die Wahrscheinlichkeit, dass nicht implementierte Nebenwirkungen die Ergebnisse verfälschen, es zu einer Zustandsraum-Explosion (*State Space Explosion*) kommt oder das *Constraint Solving* sich aufgrund seiner exponentiellen Laufzeitkomplexität nicht mehr in vertretbarer Zeit durchführen lässt [1]. Durch die Methode sind die opaken Prädikate gut getarnt – ein Parameterzugriff ist schließlich normales Verhalten in jedem Programm.

Beim Parameter-Sampling wird diese Annahme genutzt und ein zufälliger Funktionsparameter für jede zu obfuskierende Funktion ausgewählt. Um die Gefahr zu umgehen, dass dieser Parameter immer einen konstanten Wert annimmt, wurde Algorithmus 4 implementiert.

Algorithm 4 Suche nach Parameter mit extern-abgeleiteten Wert in Funktionsaufruf über DFS

```
1:  function GetSymbolicVariable(Function F ∈ Module)
2:      for all Callsite Callsite ∈ F do
3:          for all Argument Arg ∈ Callsite do
4:              Visited = HashMap()
5:              if IsVariableDerivedFromExternal(Arg, Visited) then return Arg
6:              end if
7:          end for
8:      end for
9:  end function
10:
11: function IsVariableDerivedFromExternal(Argument Arg, Hashmap Visited, Depth ∈ ℤ)
12:     if Depth > 20 ∨ Arg ∈ Visited then return false
13:     end if
14:     Visited[Arg] = true
15:     if Val is result of External Function then          ▷ Check immediate origin return true
16:     end if
17:     if Val is Internal Function Call then      ▷ Determine data sources S based on operation type
18:         S ← Return statements of the called function
19:     else if Val is Memory Read then
20:         S ← Values written to the source address (by Stores)
21:     else                                                ▷ Arithmetic, Casts, PHI inputs
22:         S ← Val.Operands
23:     end if
24:     for all Src ∈ S do                                  ▷ Recursively trace data flow
25:         if IsDerivedFromExternal(Src, Visited, Depth + 1) then return true
26:         end if
27:     end for
28:     return false
29: end function
```

5.6 Füllcode

Ohne gut getarnten Füllcode ist eine Erkennung des unwahrscheinlichen Prädikats trivial. Der Pfad, welcher die plausibelsten Anweisungen enthält ist der Richtige, unabhängig von der Qualität des opaken Prädikats. Da diese Arbeit nicht das Ziel hat, qualitativ hochwertigen Füllcode zu generieren, wird für den ungewünschten Prädikatwert der Kontrollfluss auf ein zufälligen Basisblock der zu obfuskierenden Funktion übertragen. Darf ich das so schreiben? Hierdurch erscheinen beide Pfade ausgehend vom Prädikat für einen Angreifer plausibel.

6 Implementierung

Zur Implementierung wurden drei Ansätze erwogen: die Entwicklung eines Bin2Bin-Obfuskators[6], die Implementierung in Form eines LLVM-Passes [12, 15][7] sowie die Quellcodemanipulation[8] 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.

2. **Optimierung:** Die LLVM-Toolchain enthält etablierte Optimierungs-Pässe und profitiert fortlaufend von der Arbeit zahlreicher Beiträgender. Dies ermöglicht eine nahezu optimale Kompilation obfuskierter Programme, welche sich als nützlich und sogar erforderlich in vielen Anweundungssituationen 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.

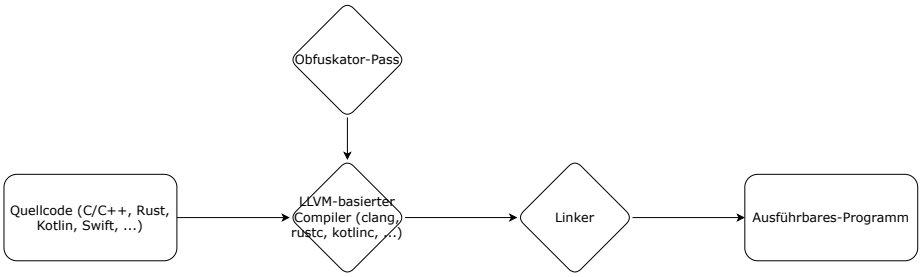


Abbildung 7: Schematische Darstellung eines LLVM-basierten Build-Prozesses mit optionalem Obfuskator-Pass.

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

Die Implementierung, Beispiele und Experimente sind unter [3] zu finden. Da in der Implementierung alle Zufahlszahlen vom selben PRNG generiert werden, kann der anfängliche Startwert (*Seed*) als Schlüssel für die Obfuskation betrachtet werden.

7 Evaluierung

7.1 Vorgehen

Für die Evaluierung wurden die weitverbreiteten Kriterien von Collberg et al. [7] verwendet:

[6] Direkte Manipulation von Assembler-Anweisungen existierender Programme.
[7] Nutzung des LLVM-Projekts, um einen sog. *Compiler-Pass* zu schreiben.
[8] durch z.B. C-Makros und das Einfügen von inline Assembler-Ausschnitten

```bash
#!/bin/bash

OPT_LVL=03
FILENAME="hello_world"
PASS_PLUGIN_DIR="Obfuscator.so"
PROB=50

clang -$OPT_LVL \
  -fpass-plugin=$PASS_PLUGIN_DIR \
  -Xclang -load -Xclang
    $PASS_PLUGIN_DIR \
  -mllvm -pop-probability=$PROB \
  ${FILENAME}.c \
  -o $FILENAME
```

(a) Bash-Skript zur Ausführung der implementierten Obfuskation auf C(++) Quellcode.

```c
__attribute__(annotate("POP"))
void foo(int x)
{
  printf("foo %i\n", x);
}
```

(b) Eine einfache zu obfuskierende Funktion.

```c
void __fastcall foo(uint64_t x)
{
  __m128d v7;
  uint64_t v8 = x;
  double v9 = x * COERCE_DOUBLE(0x4000000000000LL);
  double v10 = 35.31442506435751;
  for ( i = 0; i != 12; ++i )
  {
    __m128d v1 = *&v10;
    double v6 = (v10 - 35.29940863512881) *
    33.29686388054899;
    v1.m128d_f64[0] = v10
      - (0.0 * ((1.0 - v6) * (1.0 - v6) * (1.0 - v6))
      + 0.5018965731323943 * v6 * ((1.0 - v6) * (1.0 -
    v6))
      + 1.199789464169086 * (v6 * v6) * (1.0 - v6)
      + v6 * v6 * v6
      - v9)
      / ((0.5018965731323943 * ((1.0 - v6) * (1.0 - v6))
      + 1.395785782073383 * v6 * (1.0 - v6)
      + 1.800210535830914 * (v6 * v6))
      * 33.29686388054899);
    v7 = v1;
    v2 = _mm_cmplt_sd(0x4041AA2B238C72F8uLL, v1);
    v3 = _mm_or_pd(_mm_andn_pd(v2, v1), _mm_and_pd(v2, 0
    x4041AA2B238C72F8uLL));
    v4 = _mm_cmplt_sd(v3, 0x4041A65305AC0256uLL).
    m128d_f64[0];
    *&v10 = ~*&v4 & *&v3.m128d_f64[0] | *&v4 & 0
    x4041A65305AC0256LL;
  }
  if ( v7.m128d_f64[0] <= 35.32927409341327 )
    printf("foo %lu\n", *(&v5 - 2)); // This Basic Block
     will always be executed.
  else
  {
    // Junkcode
    // This Basic Block will never be executed.
  }
}
```

(c) Dieselbe Funktion aber mit PoP (Bernsteinpolynomgrad $n = 3$) obfuskiert. Pseudo-C wurde modifiziert aus der Dekompilation von IDA entnommen. Die Funktionen mit dem Präfix „_“ sind intrinsische Funktionen. Sie kapseln einzelne CPU-Anweiseungen.

Kriterium	Beschreibung
Stärke	Wie unverständlich ist das obfuskierte Programm für einen Angreifer?
Resilienz	Wie schwer ist eine (automatisierte) Deobfuskation?
Kosten	Wie sehr erhöht die Obfuskation die Laufzeitkosten?
Tarnung	Wie auffällig ist die Obfuskation?

Es wurde sich dabei bewusst gegen Benchmarkprogramme wie in [2][9] oder [18] entschieden: Orientiert wurde sich dabei am zur Zeit umfangreichsten Literaturreview zur (De-)Obfuskation [22]. Dieses bemängelt mangelnde Samplegröße sowie -quantität in der aktuellen Forschung. Es wurde sich daher für die LLVM Test Suite [16] entschieden. Dabei wurde sich auf die 200 größten vollständigen Anwendungen (z.B. sqlite3 & lua) und Benchmarkprogramme beschränkt. Obwohl die Test Suite eigentliche für Compileroptimierungsevaluierungen entwickelt wurde, ist sie dennoch für die Obfuskationsevaluierung geeignet, da sie durch ihre diversen Programme realistische Anwendungsszenarien der Obfuskation abbildet. Die Test Suite bietet zudem den Vorteil vorgefertigter Tests, welche die Korrektheit obfuskierter

[9] https://github.com/tum-i4/obfuscation-benchmarks

Programme prüfen. Alle Programme wurden mit clang 18.1.3 und Optimierungslevel `-O3` kompiliert und mit Ubuntu 24.04.3 LTS mit Kernel Version 5.16, einer Intel® Core™ i7-10700F CPU und 16Gb DDR4 RAM ausgeführt.

7.2 Evaluierung

7.2.1 Kosten

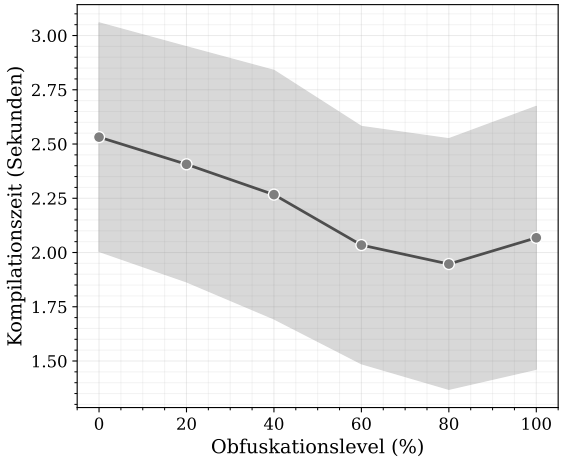


Abbildung 9: Kompilationszeit mit/ohne der Obfuskation

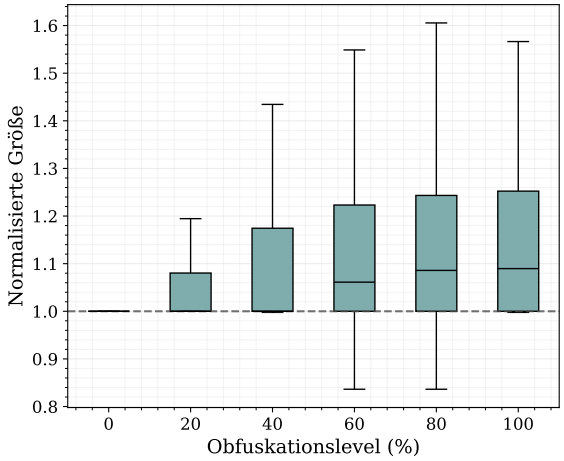


Abbildung 10: Größe vor/nach der Obfuskation

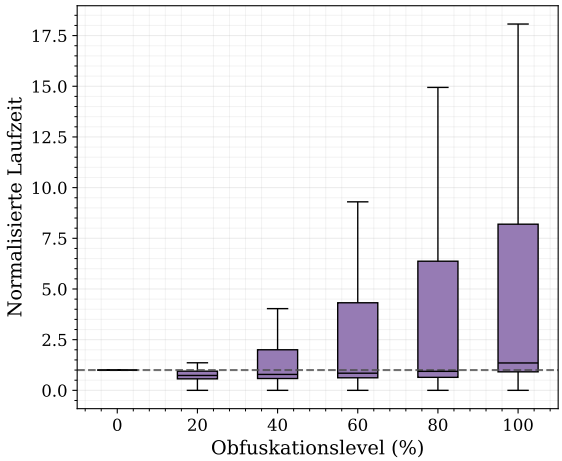


Abbildung 11: Laufzeit vor/nach der Obfuskation

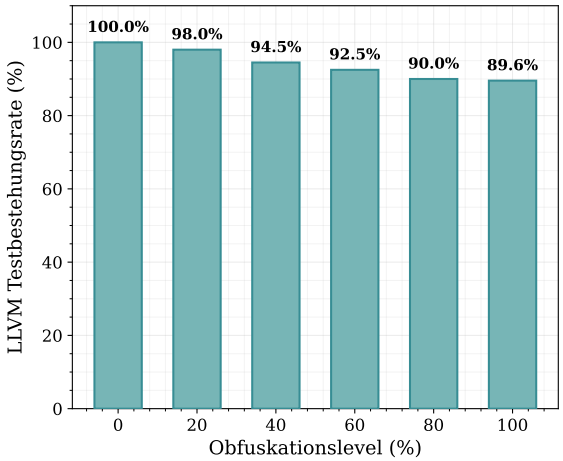


Abbildung 12: Testerfolg vor/nach der Obfuskation

Um die Obfuskationskosten zu messen wurden Programmgröße, -laufzeit, Kompilationszeit und Erfolgsquote bei den Tests der LLVM Test Suite gemessen. Insgesamt bleiben die Kosten überschaubar. Die größten Auswirkungen hatte die Obfuskation auf TODO: ... Unerwünschte Kosten in der Größe oder Laufzeit können durch ein Finetuning der Obfuskatoreinstellungen ausgeglichen werden, z.B. indem nur bestimmte Funktionen obfuskiert werden. Hervorzuheben ist, dass trotz der probabilistischen opaken Prädikate auch größere Projekte wie die GNU Core Utilities sowie SQLite fehlerfrei kompilieren und funktionierten. Hierfür wurde die Erfolgswahrscheinlichkeit $P(\mathit{TrueBB}) = 0.99$ verwendet.

TODO: Wie viel % Anwendbarkeit?

7.2.2 Stärke

Um die Stärke zu quantifizieren wurden verschiedene Komplexitätsmetriken aus der Softwareentwicklung verwendet. Konkret betrachtet wurden die zyklomatische Komplexität (McCabe-Metrik) sowie die Anzahl an Anweisungen vor und nach der Obfuskation.

7.2.3 Resilienz

Da sich die Resilienz nicht im Falle dieses Projekts nicht quantitativ messen lässt, wird sie qualitativ anhand eines Fallbeispiels untersucht. Hierfür wurden folgende Angriffsmethoden auf das einfache Programm aus Abschnitt 6 angewandt:

- Symbolische Ausführung mit Angr [20] (v.9.2.189), Triton [19] (v.1.0.0rc4) und Miasm [9] (v.0.1.5)
- Programmsynthese mittels Syntia [4] (commit 3602893)
- dynamische Analyse über mehrfache Ausführung obfuskierter Programme wie in [25] beschrieben

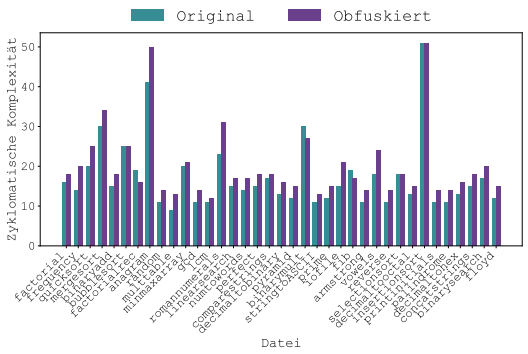


Abbildung 13: zyklomatische Komplexität (McCabe-Metrik) vor/nach der Obfuskation

Wie vermutet waren keine der symbolischen Ausführungsengines in der Lage, die Möglichkeit, dass einer der zwei Pfade nie ausgeführt wird, auszuschließen. TODO: Verhalten beschreiben (Angr fickt Computer usw.)

Eine Ausführung von Syntia ergab ...

Manuelle Angriffe und Heuristiken Die Resilienz der Obfuskation basiert somit letztendlich auf dessen Tarnung.

7.2.4 Tarnung

Um die Tarnung quantitativ zu messen wurde die Verteilung an Programmanweisungen vor und nach der Obfuskation mit Obfuskationslevel 100 gemessen.

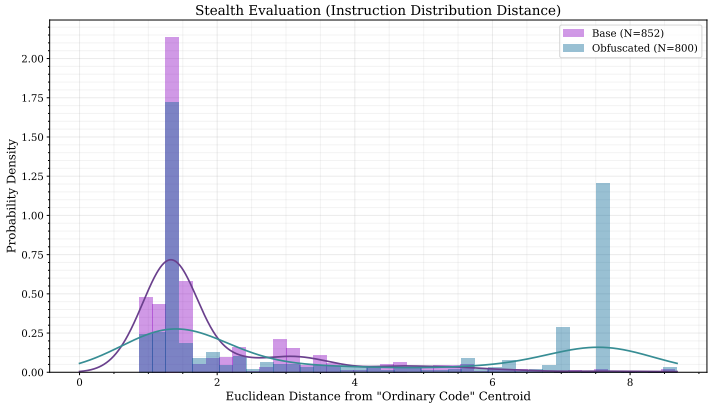
7.3 Vergleich mit existierenden Obfuskationsmethoden

8 Ergebnisdiskussion

Verbesserung durch Anwendung anderer Obfuscationsmethoden (z.B. MBAs)
Schlechtes Laufzeitverhalten relativieren

9 Fazit und Ausblick

später: Anwendung auf Ausdrücke



r[H]

Abbildung 14: Programmanweisungverteilung vor/nach der Obfuskation

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] Sebastian Banescu u. a. "Code Obfuscation against Symbolic Execution Attacks". In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. Los Angeles California USA: ACM, Dez. 2016, S. 189–200. ISBN: 978-1-4503-4771-6. DOI: 10.1145/2991079.2991114. (Besucht am 19. 12. 2025).

[3] Paul Baumgartner. *Quelltext für die Implementierung von POP*. https://github.com/sariaki/JuFo-2026. 2025. (Besucht am 29. 12. 2025).

[4] Tim Blazytko u. a. "Syntia: Synthesizing the Semantics of Obfuscated Code". In: ().

[5] Adrian Colesa, Radu Tudoran und Sebastian Banescu. "Software Random Number Generation Based on Race Conditions". In: *2008 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 2008, S. 439–444. DOI: 10.1109/SYNASC.2008.36.

[6] Christian Collberg. *The Tigress C Obfuscator*. https://tigress.wtf/. 2025. (Besucht am 14. 10. 2025).

[7] 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).

[8] 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).

[9] F. Desclaux und C. Mougey. *Miasm2: Reverse engineering framework*. Slides, Black Hat / conference presentation. https://i.blackhat.com/us-18/Wed-August-8/us-18-DesclauxMougey-Miasm-Reverse-Engineering-Framework.pdf, accessed: <access-date>. Aug. 2018.

[10] 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.

[11] 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.

[12] 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.

[13] 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).

[14] "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).

[15] LLVM Project. *The LLVM Compiler Infrastructure*. https://llvm.org/. Version 20.1.4. 2003–2025. (Besucht am 18. 08. 2025).

[16] LLVM Project (llvm-test-suite Contributer). *llvm-test-suite*. https://github.com/llvm/llvm-test-suite. 2025.

[17] Toshio Ogiso u. a. "Software Obfuscation on a Theoretical Basis and Its Implementation". In: (2003).

[18] Mathilde Ollivier u. a. "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*. San Juan Puerto Rico USA: ACM, Dez. 2019, S. 177–189. ISBN: 978-1-4503-7628-0. DOI: 10.1145/3359789.3359812. (Besucht am 22. 12. 2025).

[19] Florent Saudel und Jonathan Salwan. "Triton: A Dynamic Symbolic Execution Framework". In: *Symposium sur la sécurité des technologies de l'information et des communications*. SSTIC. Rennes, France, Juni 2015, S. 31–54.

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

[21] 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).

[22] Bjorn De Sutter u. a. *Evaluation Methodologies in Software Protection Research*. Apr. 2024. DOI: 10.48550/arXiv.2307.07300. arXiv: 2307.07300 [cs]. (Besucht am 19. 12. 2025).

[23] 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.

[24] 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).

[25] Lukas Zobernig, Steven D. Galbraith und Giovanni Russello. "When Are Opaque Predicates Useful?" In: *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE). Rotorua, New Zealand: IEEE, Aug. 2019, S. 168–175. DOI: 10.1109/trustcom/bigdatase.2019.00031. URL: https://ieeexplore.ieee.org/document/8887369/ (besucht am 17. 07. 2025).

Abbildungsverzeichnis

1 Kontrollflussgraph einer einfachen Funktion mit opakem Prädikat. Abbildung aus der Disassembly des Spiels *Overwatch* mittels IDA entnommen. 2
2 Ausschnitt des Kontrollflussgraphen einer Funktion mit vielen opaken Prädikaten. Durch die vielen opaken Prädikate wird die Funktion unübersichtlich und eine Analyse aufgrund der Unklarheit tatsächlich ausführbarer Pfade erschwert. Abbildung aus der Disassembly des Spiels *Overwatch* mittels IDA entnommen. 2
3 Beispiele publizierter opaker Prädikate zur Verhinderung symbolischer Ausführung. . . . 4
4 Konzeptuelles Framework zur Erkennung opaker Prädikate mit symbolischer Ausführung [24] 4
5 Zufällige Zahlen y_i werden von einer Gleichverteilung $Unif(0, 1)$ generiert. Jeder zufällige y-Wert wird über die inverse kumulative Verteilungsfunktion der Exponentialverteilung $F^{-1}(x)$, einem x-Wert zugeordnet. Wie bei einer Exponentialverteilung sammeln sich hierdurch die $x = F^{-1}(x)$-Werte um 0. 7
6 Zufällige Bernsteinpolynome fungieren als verschiedene kumulative Verteilungsfunktionen für die probabilistischen opaken Prädikate. 8
7 Schematische Darstellung eines LLVM-basierten Build-Prozesses mit optionalem Obfuskator-Pass. 11
9 Kompilationszeit mit/ohne der Obfuskation 13
10 Größe vor/nach der Obfuskation 13
11 Laufzeit vor/nach der Obfuskation 13
12 Testerfolg vor/nach der Obfuskation 13
13 zyklomatische Komplexität (McCabe-Metrik) vor/nach der Obfuskation 14
14 Programmanweisungverteilung vor/nach der Obfuskation . . . . . A