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

Projektüberblick

1 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 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. 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 automatisierte Deobfuskation mit aktuellen Methoden unmöglich machen?* Die Relevanz dieser Forschungsfrage ergibt sich aus dem fortdauernden Kampf zwischen Obfuskation und Deobfuskation sowie der Effektivität von symbolischer Ausführung gegen opake Prädikate.

Die Kernbeiträge dieser Arbeit hierzu sind folgende:

- Es wird das neue Konzept probabilistischer opaker Prädikate vorgestellt. Es handelt sich dabei um opake Prädikate, welche durch die Nutzung von Wahrscheinlichkeitsverteilungen und Pseudozufallsvariablen verschiedene Angriffsmethoden verhindern.
- Es wird ein Algorithmus geliefert, um die neue Art der opaken Prädikate mit verschiedenen Wahrscheinlichkeitsverteilungen zu generieren. Der Algorithmus wird implementiert und erfolgreich evaluiert.
- Dabei wird ein Problem bei der Generierung symbolischer Variablen über Funktionsparameter angesprochen und Algorithmus 4 als Lösung präsentiert. Nach bester Kenntnis des Autors handelt es sich hierbei um ein neuen undokumentierten Angriffsvektor.

In dieser Arbeit wird sich bewusst auf die Erzeugung der opaken Prädikate selbst beschränkt. Die Relevanz von Füllcode (sog. *Junkcode*) ist unabhängig von den dargestellten Idee und wird daher nicht behandelt. Zunächst werden Obfuskation, opake Prädikate und symbolische Ausführung formal definiert (Abschnitt 2). Durch einen Überblick über den aktuellen Stand der Forschung (Abschnitt 3) wird die Notwendigkeit dieser Arbeit hergeleitet und ein Angreifermodell festgelegt (Abschnitt 4.1). In Abschnitt 4.2-3 wird die Idee propabilistischer opaker Prädikate vorgestellt und definiert. Darauf werden verschiedene Methoden zur zufälligen Wahrscheinlichkeitsverteilungsgenerierung (Abschnitt 4.4) sowie zur Generierung von Pseudozufallszahlen (Abschnitt 4.5) vorgestellt und abgewogen. Dabei wird das bis jetzt nicht untersuchte Problem der Aufruf-Invarianz für die Generierung symbolischer Variablen über Funktionsparameter angesprochen und über Algorithmus 4 gelöst. Implementiert wird die Idee in Form eines LLVM-Passes (Abschnitt 5). Eine Evaluierung anhand der Kriterien Collbergs et al. [8] (Abschnitt 6) zeigt, dass probabilistische opake Prädikate resilient gegenüber symbolischer Ausführung und weiteren Methoden sind, ohne dabei unvertretbare Kosten aufzuweisen.

Diese Arbeit geht von einem informatischen Grundwissen an Assembler, Compilern sowie Programmanalysemethoden aus. 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 Programmanweisungen modifiziert nimmt an, dass diese in *Single Static Asignment*-Form definiert sind.

2 Theoretische Grundlagen

2.1 Obfuskation

Collberg et al. [8] definieren Obfuskation wie folgt:

Definition 2.1 (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.

Obfuskation zielt darauf ab, die Komplexität eines Programmes so zu erhöhen, dass dessen interne Logik für einen Angreifer nur schwer verständlich ist. 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*.

2.2 Opake Prädikate

Die folgenden Definitionen sind aus [24] sowie vom Pionierwerk [9] modifiziert übernommen. Es wird sich auf hierbei auf invariante opake Prädikate beschränkt.

Definition 2.2 (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 [9].

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

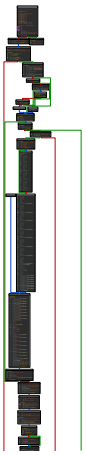

Abbildung 2: Ausschnitt des Kontrollflussgraphen einer Funktion mit vielen opaken Prädikaten. Durch die vielen opaken Prädikate wird die Funktion unübersichtlich und eine Analyse aufgrund der Unklarheit tatsächlich ausführbarer Pfade erschwert. Abbildung aus der Disassembly des Spiels *Overwatch* mittels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

Opake Prädikate werden in der Softwareobfuskation eingesetzt, um ein Verständnis über den Kontrollfluss des Programms zu behindern [9, 24]. 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 [9]. 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 [13] 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.*

2.3 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 [2]. Die 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 [2].

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

Hierdurch können schließlich über den Constraint-Solver allgemeine Aussagen über die Erreichbarkeit bestimmter Pfade oder Variablenwerte getroffen werden [2]. Ist eine Pfadformel erfüllbar, kann der Solver zudem konkrete Eingebewerte hierfür liefern [2]. 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 [2]. Zur Lösung wurden verschiedene Ansätze (z.B. *Concolic Execution*) [2] entwickelt. Aufgrund der Fülle an Informationen und der geringen Relevanz für die in dieser Arbeit dargestellten Abwehrmethodik, wird auf ihre Darstellung verzichtet..

3 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

[1] D.h, dass der wirkiche Pfad, welcher von einem opaken Prädikat verschleiert wird, nicht einfach erkannt werden kann
[2] Es handelt sich meist um einen SMT-Solver.

und Herausforderungen zu verdeutlichen. Die geschieht anhand der Kriterien aus Abschnitt 6.1.

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 [26].

Existierende Methoden lassen sich im Wesentlichen in zwei Kategorien einteilen:

1. Opake Prädikate, welche Implementierungsschwächen[3] existierender symbolischer Ausführungsengines angreifen.

   Hierunter fällt die Nutzung von *Exceptions* [15], Anweisungen [25] oder Funktionen [7], welche nicht durch die symbolischen Ausführungsengines modelliert werden. Eine Befragung von Audrey Dutcher, einer der Entwicklerinnen vom Programmanalyseframework *Angr* [22] ergab: drei der vier in [25] dargestellten Methoden können nun von dem symbolischen Ausführungsengine von *Angr* problemlos symbolisch ausgeführt werden [4].Eine Deobfuskation weiterer Methoden in dieser Kategorie liegt also alleine in der Verbesserung existierender symbolischer Ausführungsengines. Eine Deobfuskation ist in gewisser Frage nur eine Frage der Zeit.

2. Opake Prädikate, welche fundamentale Grenzen der *Constraint Solver* angreifen.

   Diese Nutzen entweder ungelöste Probleme in der Mathematik oder NP-schwere Probleme der Informatik. So erstellte [16] z.B. opake Prädikate, welche auf der unbewiesenen Vermutung basieren, dass die Collatz Folge unabhängig vom Startwert immer den Zyklus $4, 2, 1$ erreicht. Da Mathematiker noch nicht die Werkzeuge haben, dies zu beweisen, ist die symbolische Ausführung auch nicht in der Lage zu beweisen, dass das Programm weiterläuft und nicht in eine Endlosschleife gerät. In [19] wird alternativ ein NP-schweres Problem mit Pointerarithmetik konstruiert. In beiden Fällen sind die opaken Prädikate zwar beweisbar sicher, allerdings einfach erkennbar und mit hohen Laufzeitkosten verbunden.

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

4 Ansatz

4.1 Angreifermodell

[3] bzw. Heuristikschwächen.

[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 [25] über Dateisystem-Operationen implementiert: Ausführbar.
(d) Threads: Noch nicht implementiert.

Diese Arbeit geht aufgrund der Ähnlichkeit behandelter Thematik von einem *Man-at-the-End* (MATE) Angreifermodell aus, aufbauend auf [25, 26]. 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. 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. Mögliche dynamischer Analysemethoden werden in Abschnitt 7 angeführt und diskutiert.

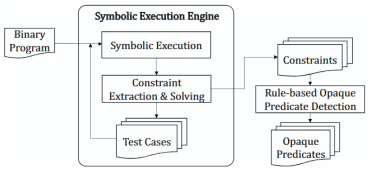

Abbildung 3: Konzeptuelles Framework zur Erkennung opaker Prädikate mit symbolischer Ausführung. Abbildung aus [25] übernommen.

4.2 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 4.1 (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. Formaler mit ϵ wie Barak et al.?

Definition 4.2. 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 4.3. $\mathcal{P}_{prob}$ 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.

4.3 Algorithmus

Für jedes zu generierendes opakes Prädikat wird im Programm ein Pseudozufallsvariable U generiert. Verschiedene Methoden hierfür werden in Abschnitt 4.5 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 4.4 genauer vorgestellt. Mit dieser transformierten Zufallsvariable können nun wahrscheinliche bzw. unwahrscheinliche probabilistische Prädikate erstellt werden.

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

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

4.4 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 [12]:

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

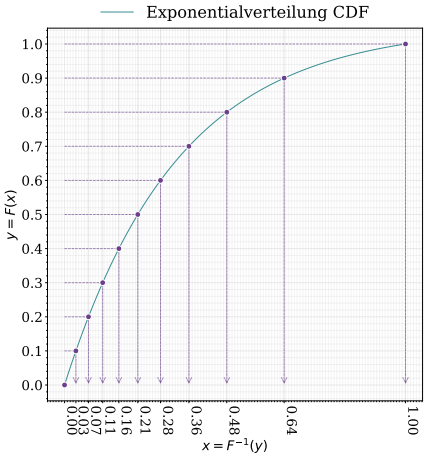

Abbildung 4: Zufällige Zahlen y_i werden von einer Gleichverteilung $Unif(0, 1)$ generiert. Jeder zufällige y-Wert wird über die inverse kumulative Verteilungsfunktion der Exponentialverteilung $F^{-1}(x)$, einem x-Wert zugeordnet. Wie bei einer Exponentialverteilung sammeln sich hierdurch die $x = F^{-1}(x)$-Werte um 0.

Diese hat den Vorteil, dass sich verschiedene andere Verteilungen (z.B. Chi-Quadrat-, Erlang und Expo-

nentialverteilung) 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.
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.

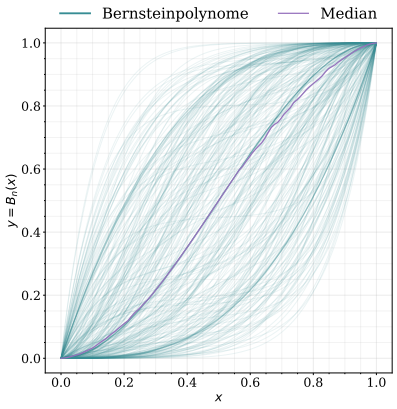

Abbildung 5: Zufällige Bernsteinpolynome fungieren als verschiedene kumulative Verteilungsfunktionen für die probabilistischen opaken Prädikate.

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

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

Mantisse korrekt zu bestimmen. Um Sicherheitsgarantien zu liefern wurde der Algorithmus mit $k = 8$ 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.¸

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

4.5 Generierung von Pseudozufallsvariablen

Damit der vorgestellte Ansatz funktionieren kann, bedarf er einer gleichverteilten (Pseudo-)Zufallszahl, welche auf dem Einheitsintervall liegt und zugleich als unbestimmte symbolische Variable ohne konkreten Wert von symbolischen Ausführungsengines betrachtet wird. Hierfür wird für jede zu obfuskierende Funktion ein zufälliger Funktionsparameter ausgewählt und auf dem Einheitsintervall $[0; 1]$ abgebildet. Hierdurch sind die opaken Prädikate gut getarnt – ein Parameterzugriff ist schließlich normales Verhalten in jedem Programm. Die einzige Gefahr ist dabei, dass der gewählte Parameter immer konstante Werte annimmt. Ein Angreifer könnte in diesem Fall alle Werte dieses Parameters in Funktionsausrufen sammeln und die Funktion mit ihnen ausführen. Sind für jeden Parameterwert die ausgeführten Basisblöcke gleich, so handelt es sich bei den nicht ausgeführten Basisblöcken um „Junkcode“. Prädikate, welche auf diese Basisblöcke verweisen sind folglich opak und können gelöscht werden. **Diese Vulnerabilität wurde in**

keiner der gelesenen Arbeiten zum Thema bis jetzt angesprochen. Algorithmus 4 bietet hierfür eine Lösung.

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

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

4.6 Füllcode

TODO: entfernen? 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 unwahrscheinlicheren Prädikatwert der Kontrollfluss auf ein zufälligen Basisblock der zu obfuskierenden Funktion übertragen. Für den Machbarkeitsnachweis dieser Arbeit genügt dies. Eine für den Praxisgebrauch bestimmte Implementierung müsste dies mit den dargelegten Methoden anderer Arbeiten ergänzen.

5 Implementierung

Zur Implementierung wurden drei Ansätze erwogen: die Entwicklung eines Bin2Bin-Obfuskators[6], die Implementierung in Form eines LLVM-Passes [14, 17][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.

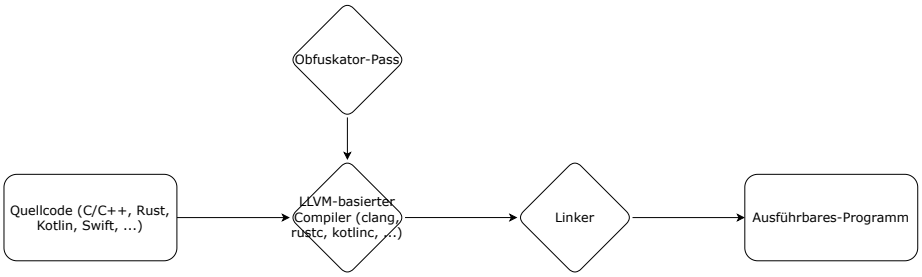

Abbildung 6: Schematische Darstellung eines LLVM-basierten Build-Prozesses mit optionalem Obfuskator-Pass.

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 [5] 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.

6 Evaluierung

6.1 Vorgehen

Für die Evaluierung wurden die weitverbreiteten Kriterien von Collberg et al. [8] verwendet: TODO: Reihenfolge anpassen!!!

[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=O3
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) *
    33.296863880548899;
    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)) + 1.1997894641690086 * (v6 * v6)
    * (1.0 - v6) + v6 * v6 * v6 - v9) /
    ((0.5018965731323943 * ((1.0 - v6) * (1.0 - v6)) +
    1.3957857820733833 * v6 * (1.0 - v6) +
    1.8002105358309144 * (v6 * v6)) * 33.296863880548899);
    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 Unterstrich als 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 bewusst gegen die einfachen Benchmarkprogramme wie in [3, 4] oder [20] entschieden. Orientiert wurde sich dabei am zur Zeit umfangreichsten Literaturreview zur (De-)Obfuskation [23]. Dieses bemängelt mangelnde Samplegröße sowie -quantität in der aktuellen Obfuskationsforschung. Es wurde sich daher für die LLVM Test Suite [18] entschieden. Dabei wurde sich auf die 200 größten vollständigen Anwendungen (darunter z.B. SQLite & Lua) und Benchmarkprogramme beschränkt. Obwohl die Test Suite eigentliche für Compilerevaluierungen 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 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. Es wurden dabei $k = 8$ Newton-Raphson-Iterationen gewählt, $n = 4$ als durchschnittlichen Bernsteinpolynomgrad sowie Erfolgswahrscheinlichkeit $P(RealBB) = 0.99$ verwendet. Jede Funktion bekommt dabei maximal ein probabilistisches opakes Prädikat.

6.2 Evaluierung

6.2.1 Kosten

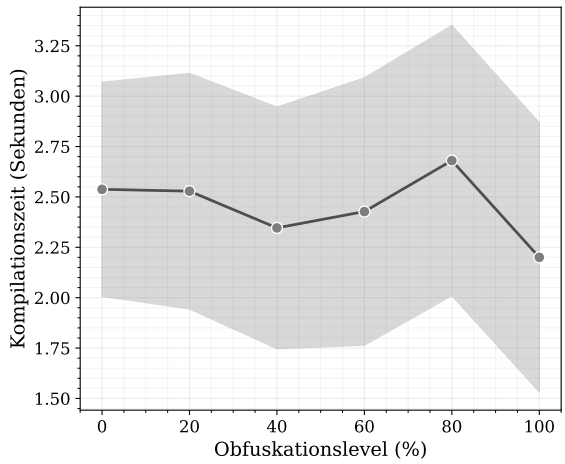

Abbildung 8: Kompilationszeit in Abhängigkeit vom Obfuskationslevel

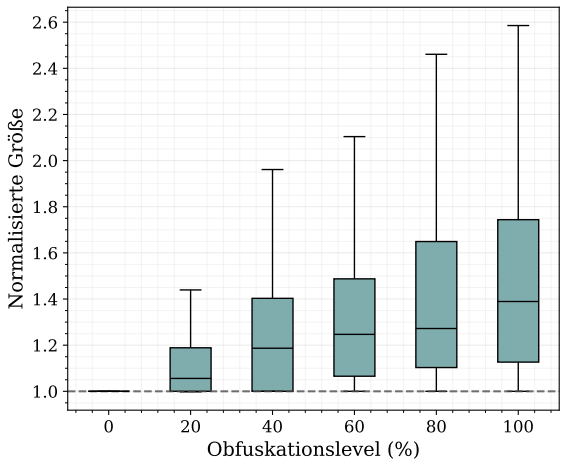

Abbildung 9: Programmgröße in Abhängigkeit vom Obfuskationslevel

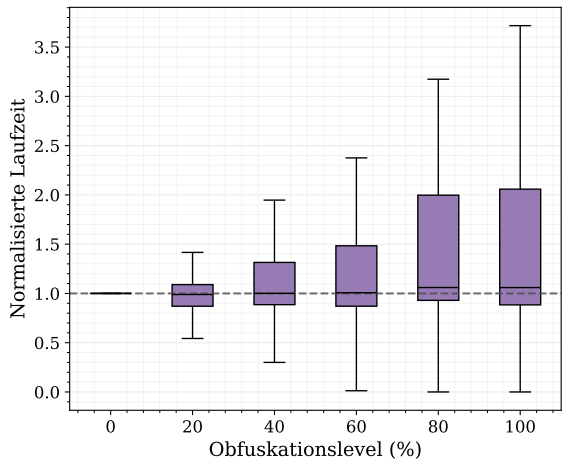

Abbildung 10: Laufzeit in Abhängigkeit vom Obfuskationslevel

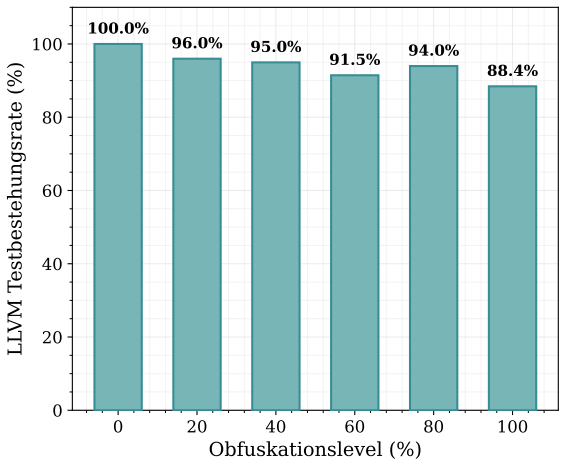

Abbildung 11: Testerfolg in Abhängigkeit vom Obfuskationslevel

Um die Obfuskationskosten zu messen wurden die durchschnittliche Programmgröße, -laufzeit, Kompilationszeit und Erfolgsquote bei den Programmen der LLVM Test Suite erhoben.

Die größten Auswirkungen hatte die Obfuskation auf die Programmlaufzeit. Die sich in der Größe oder Laufzeit ergebenden Kosten lassen sich allerdings durch ein Finetuning der Obfuskatoreinstellungen vermindern, z.B. indem nur bestimmte Funktionen obfuskiert werden. Hervorzuheben ist, dass trotz der probabilistischen opaken Prädikate auch größere Projekte wie die SQLite problemlos kompilieren und fehlerfrei funktionierten. Der Abnehmende Testerfolg in Abb. 11 liegt an den für die Tests als Fehler geltenden unerwarteten Gleitkommazahl-Flagänderungen. Insgesamt sind die Kosten je nach Programm/Anwendungsfall und Obfuskationslevel praxisgerecht.

6.2.2 Stärke

Wie in [25] angemerkt wurde, ergibt eine Quantifizierung von Stärke bei der Evaluierung neuer opaker Prädikate wenig Sinn. Relevante Metriken wie die zyklomatische Komplexität (McCabe-Metrik) erhöhen

sich beliebig mit der Häufigkeit opaker Prädikate und nicht durch ihre Art.

Auf qualitativer Ebene lässt sich anmerken, dass probabilistische opaken Prädikate indirekt auch die Schwachstellen der Werkzeuge heutiger Angreifer angreifen. Konkret sind moderne Decompiler wie IDA/Binary Ninja/Ghidra noch nicht in der Lage, bestimmte SIMD (*Single Instruction, Multiple Data*)-Anweisungen zu dekompilieren. Dies lässt sich an Abb. 7c erkennen, wo IDA die unbekannten Anweisungen durch intrinsische Funktionen darstellt anstelle von verständlicherem Pseudo-C. Experimente mit *Decompiler Explorer* [1] zeigen, dass keine der aktuellen State-of-the-Art-(SOTA)-Decompiler bis auf Ghidra in der Lage ist, die SIMD Anweisungen in Pseudo-C zu dekompilieren. Durch eine Feinjustierung der clang Kommandozeilenoptionen nutzten die kompilierten Programme AVX2 Anweisungen. Keine der SOTA-Decompiler konnten diese in Pseudo-C dekompilieren. Diese Limitation bedeutet für Angreifer, dass eine Verständnis der Obfuskation (und somit auch der Programmlogik) mehr Zeit, Aufwand und vor allem ein Verständnis von Assembler erfordert. Die Hürde zum erfolgreichen Reverse-Engineering wurde somit erhöht.

6.2.3 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 5 sowie auf TODO angewandt:

- Symbolische Ausführung mit Angr [22] (v.9.2.189), Triton [21] (v.1.0.0rc4) und Miasm [11] (v.0.1.5)
- Programmsynthese mittels Syntia [6] (commit 3602893) TODO: Xyntia nutzen!
- Probabilistische Analyse über mehrfache Ausführung obfuskierter Programme mit zufälligen Werten [10]

Wie vermutet waren keine der symbolischen Ausführungsengines in der Lage, die Möglichkeit, dass einer der zwei Pfade nie ausgeführt wird, auszuschließen. Alle drei symbolischen Ausführungsengines waren nicht in der Lage, innerhalb von 8 Stunden die Constraints des unwahrscheinlichen Pfades zu extrahieren und hängten.

Eine Ausführung von Syntia entfernte nicht nur die Obfuskation, sondern auch wesentlichen Programmcode, sodass das Programm nicht mehr dasselbe Laufzeitverhalten in allen Situationen hatte.

Wie in [10] empirisch untersucht, markiert eine probabilistische Analyse auch hier fälschlicherweise 20 – 40% echter Prädikate als opak. Dies ist damit zu begründen, dass auch viele Prädikate der unobfuskierten Programmlogik besonders häufig gleiche Werte annehmen.

Trotzdem ist eine Deobfuskation möglich. Erkennt ein Angreifer ein Prädikat als opak, so muss er nur über dynamischer Ausführung den korrekten Pfad bestimmen. Die Resilienz der Obfuskation basiert somit auf dessen Tarnung. Der Autor hält eine automatisierte Erkennung mittels Pattern-Matching aufgrund der generalisierten Implementierung sowie der Messwerte aus Abschnitt 6.2.4 für unwahrscheinlich. Dies bestätigt sich auch empirisch. Es konnten keine Patterns erstellt werden, welche alle probabilistisch opaken Prädikate abdeckten. Nicht gemessen wurde die Möglichkeit, über Machine Learning Methoden wie [24], die opaken Prädikate wiederzuerkennen oder sogar zu deobfuskieren.

Zudem ist eine Anwendung von an die Obfuskationsmethode angepassten Heuristiken möglich. Ein Angreifer könnte z.B. alle Prädikate mit symbolischen Ausführungsengine ausführen und die benötigte

Zeit für die Extraktion der Constraints beider Pfade messen. Dauert dies für einen Pfad länger als eine festgelegte konstante Zeit, so handelt es sich mit hoher Wahrscheinlichkeit um ein probabilistisch opakes Prädikat. Trotzdem hat die Methode denselben Nachteil wie die probabilistische Analyse: ihre Korrektheit ist nie garantiert.

6.2.4 Tarnung

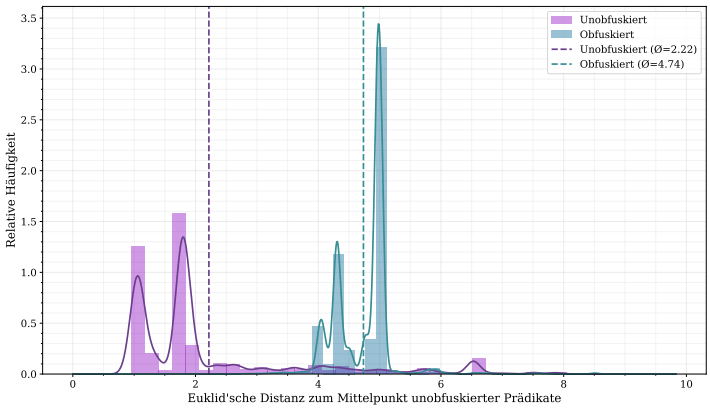

Abbildung 12: Programmanweisungverteilung vor/nach der Obfuskation

Kategorie	**Anweisungen**
Arithmetik	`imul, inc, sub, add, idiv, divsd, sbb, subpd, addpd, mulpd, divpd, maxpd, minpd, sqrtpd`
Logik	`and, sar, xor, test, shr, shl, or, xorps, xorpd, orpd, andpd`
Datenübertragung	`movaps, movsd, movabs, movzx, mov, movss, movsx, movsxd, stosd, movapd, movupd, unpcklpd, unpckhpd, movhpd, movlpd`
Datenbreite/-konversion	`cvtss2sd, cvtsi2sd, cvtsd2ss, cqo, cdq, cvttsd2si`
Zeigerarithmetik	`lea`
Vergleich	`cmp, ucomisd, comisd`
Sprünge	`jle, jne, jge, jae, jl, je, jg, jp, ja, jbe, jno, jmp`
Stapel	`pop, push, call, ret`
Boolesch	`setge, setne, setg, seta, setb, setl, sete`
Sonstige	`nop`

Tabelle 1: 74 Assembler-Anweisungen in 10 Kategorien sortiert.

Um die Tarnung quantitativ zu messen wurde wie bei [25] vorgegangen. Hierfür wurden aus den insgesamt 200 Programmen n =TODO jeweils unobfuskierte sowie obfuskierte Prädikate zufällig gewählt. Gemäß Tabelle 1 wurden die letzten zehn Anweisungen vor der bedingten Sprungsanweisung (inkl. der Sprunganweisung selbst) kategorisiert. Hierdurch kann jedes Prädikat i als 10-dimensionaler Vektor $\vec{x}_i$ dargestellt werden, wobei jede Dimension der absoluten Häufigkeit einer Anweisungskategorie in den letzten 10 Anweisungen vor der Sprunganweisung entspricht. Um den Unterschied zwischen obfuskierten und unobfuskierten Prädikaten zu messen wurde zuerst der Mittelpunkt $\vec{m} = \frac{\sum_{i=1}^{n} \vec{x}_i}{n}$ aller unobfuskierten Prädikate berechnet. Die durchschnittliche Euklid'sche Distanz der obfuskierten Prädikate zum Mittelpunkt $\vec{m}$ entspricht ihrer Ähnlichkeit und somit auch Tarnung. Je geringer der Unterschied ist, desto höher ist die Tarnung. Wie Abb. 12 zu entnehmen ist, TODO!!! Den größten Unterschied gab es bei Programmen mit nur wenigen Gleitkommazahlberechnungen. Die Tarnung lässt sich hier erhöhen durch eine

Kombination mit anderen Obfuskationsmethoden, z.B. dem Einfügen von „Dead Code“/„Junk Code“ mit vielen Gleitkommazahloperationen.

7 Fazit

In der vorliegenden Arbeit wurde das neue Softwareobfuskationskonzept der probabilistischen opaken Prädikate eingeführt und in einer Implementierung in Form eines LLVM-Passes ein Machbarkeitsnachweis geliefert. Empirische Untersuchungen bestätigen, dass die vorgestellte Idee hinsichtlich ihrer Kosten, Resilienz, Stärke und Tarnung je nach Anwendungsfall und Kombination mit anderen Obfuskationsmethoden praxisgerecht ist. Dabei konnte nebenbei ein undokumentierter Angriffsvektor erkannt und eine Lösung dafür geboten werden.

Die Resilient probabilistischer opaker Prädikate gegen symbolische Ausführung wurde theoretisch und praktisch begründet. Praktische Untersuchungen zeigen zudem eine Resilienz gegen weitere Angriffsmethoden, wie die Programmsynthese, eine probabilistische Analyse sowie Pattern-Matching. Die Forschungsfrage ist somit bestätigt.

Offen bleibt die Resilienz gegenüber Machine Learning basierten Deobfuskationattacken wie [24]. Gegenstand einer weiterführenden Auseinandersetzung sollte die Anwendbarkeit der hier dargelegten Prinzipien der probabilistischen Obfuskation auf andere Bereiche der Softwareobfuskation sein.

Danksagung

Ich danke meinem schulischen Projektbetreuer, Herrn Dr. Arndt Latußeck für die Überprüfung meiner Forschungsfrage, wichtige Hinweise zur Gliederung, Vorschlägen zum Vorgehen bei der Evaluierung sowie der fachlichen Überprüfung der vorliegenden Arbeit.

Zudem danke ich meiner Mutter, Claudia Baumgartner-Bardubitzki für ihr sprachliches/formales Korrektorat.

Des Weiteren danke ich allen Entwicklern verschiedener Open-Source Softwarebibliotheken und -Anwendungen. Ohne diese wäre eine Ausarbeitung in der begrenzten Zeit in aktueller Form unmöglich gewesen.

Literaturverzeichnis

[1] Vector 35. *Quelltext von Decompiler Explorer*. https://github.com/decompiler-explorer/decompiler-explorer. 2025. (Besucht am 29. 12. 2025).

[2] 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).

[3] 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).

[4] Sebastian Banescu u. a. *Quellcode für Obfuscation Benchmarks*. https://github.com/tum-i4/obfuscation-benchmarks. 2025. (Besucht am 29. 12. 2025).

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

[6] Tim Blazytko u. a. "Syntia: Synthesizing the semantics of obfuscated code". In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, S. 643–659.

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

[8] 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).

[9] 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).

[10] Mila Dalla Preda u. a. "Opaque Predicates Detection by Abstract Interpretation". In: *Algebraic Methodology and Software Technology*. Hrsg. von David Hutchison u. a. Bd. 4019. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, S. 81–95. ISBN: 978-3-540-35633-2 978-3-540-35636-3. DOI: 10.1007/11784180_9. (Besucht am 01. 01. 2026).

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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).

[16] “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).

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

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

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

[20] 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).

[21] 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.

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

[23] 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).

[24] 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.

[25] 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).

[26] 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