UNIVERSITY OF APPLIED SCIENCES

Seminararbeit IT-Sicherheit

WS 2017/2018

Seminar IT-Sicherheit

Code obfuscation against symbolic execution attacks

Autor: Henrik Peters

Betreuer: Prof. Dr. Gerd Beuster
Datum: 18. Februar 2018

Inhaltsverzeichnis

Grundlagen

1.1 Prinzip source code obfuscation

Software Obfuscation wird in der Softwaretechnik verwendet, um Quelltext oder Maschinencode absichtlich schwer verständlich zu machen. Damit soll der Aufwand für Reverse Engineering so weit wie möglich erhöht werden. Im besten Fall ist der Aufwand so hoch, dass sich ein Reverse Engineering praktisch nicht lohnt oder nicht mehr sinnvoll durchführbar ist. Obfuskation wird auch eingesetzt, um den Diebstahl von Programmteilen, Algorithmen oder geheimen Daten zu erschweren. Schadsoftware verwendet sehr häufig eine Obfuskation, um die eigene Funktionalität zu verschleiern und Analysen zu erschweren.

In dieser Ausarbeitung geht es allerdings nur um den Bereich der Source Code Obfuscation, d.h. eine Transformation des Quelltexts vor dem Kompilieren. Als Obfuscator kommt tigress zum Einsatz (http://tigress.cs.arizona.edu/). Tigress bietet verschiedene Transformationen für C Quelltext, die beliebig kombiniert bzw. wiederholt werden können. Das Prinzip wird durch die folgende Grafik verdeutlicht:

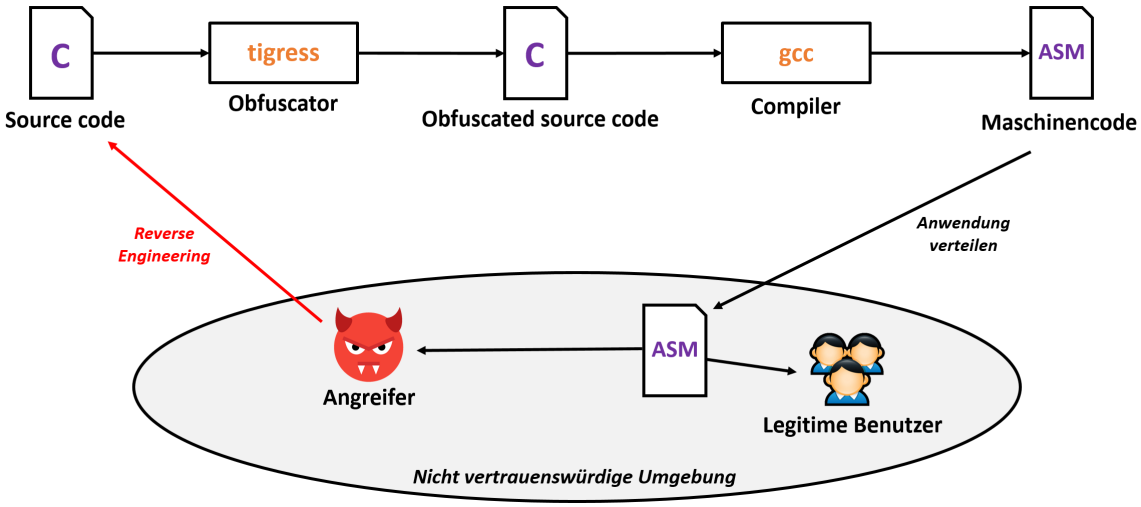


Abbildung 1.1: Prinzip source code obfuscation

Es ist schwierig die Effektivität einer Obfuscation systematisch zu erfassen und damit zu bewerten. Diese Ausarbeitung basiert im wesentlichen auf dem Paper "Code Obfuscation Against Symbolic Execution Attacks" [2]. Dieser Artikel versucht die Effektivität von Obfuskation gegen automatisierte Angriffe besser einschätzen zu können. Dies erfolgt empirisch auf der Grundlage von 5000 verschiedenen (eher kleinen) C-Programmen. Dabei wird ein besonderer Fokus auf symbolisch basierte Analysen gelegt. Ein Problem bei der Verwendung von Obfuscation ist, dass oft nicht klar definiert werden kann, inwieweit eine bestimmte Transformation gegen verschiedenste Angriffsarten schützen kann.

1.2 Motivation für Reverse Engineering

Die Ziele des Reverse Engineering zu benennen ist wichtig, um den Nutzen von konkreten Transformation genauer bestimmen zu können. Allgemein können diese Ziele jedoch sehr verschieden sein. Die in der Praxis verfolgten Ziele sollten sich jedoch in eine der folgenden Kategorien [9] einteilen lassen:

Daten im Speicher lokalisieren
In einem Programm eingebettete Daten werden nur von einigen Funktionen bzw. Programmteilen verwendet. Da diese Daten durch Obfuskation verstreut/versteckt sein können, muss zunächst das Programm einen Codeteil ausführen, der diese Daten verwendet, um die Daten logisch sinnvoll extrahieren zu können. Daten die für Angreifer in der Praxis relevant sein könnten sind z.B. Sicherheitsrichtlinien, Lizenzschlüssel, Zertifikate, Zugangsdaten, Passwörter oder Konfigurationsdateien.

Programmfunktionalitäten lokalisieren
Einstiegspunkte von Funktionen zu finden kann ebenfalls ein Ziel sein. Um die Funktionalität eines Programms zur Laufzeit zu untersuchen, muss zunächst eine passende Eingabe gefunden werden, unter der das Programm die entsprechende Funktionalität ausführt. Das Auffinden von solchen Funktionen kann nützlich sein um z.B. Integritätschecks zu umgehen, manuelles Reverse Engineering von kleineren Codeteilen zu ermöglichen oder zum Untersuchen von potentieller Schadsoftware.

Extrahieren von Code-Fragmenten
Wenn die Komponenten des Programms nur lose gekoppelt sind, d.h. wenig ineinander verwickelte Abhängigkeiten haben, können Teile eines Programms sehr leicht extrahiert werden. Dies ermöglicht es Codeteile in einer anderen Anwendung wiederzuverwenden, über die der Angreifer die Kontrolle hat. Dazu ist kein tieferes Verständnis für die eigentliche Funktionalität notwendig. Dies könnte beispielsweise der Fall sein, wenn ein Unternehmen eine Technik der Konkurrenz verwenden will, ohne aber die dafür juristische Berechtigung zu besitzen.

Verstehen des ursprünglichen Programms
In diesem Fall ist das Ziel, das ursprüngliche Programm (ohne Obfuskation) vollständig zu rekonstruieren und damit komplett zu verstehen. Dies ist deutlich aufwendiger als nur Teile des Programms zu finden oder zu extrahieren. Eine vollständige Automatisierung ist hier in der Regel nicht möglich, d.h. es muss auch menschlicher Aufwand investiert werden. Motivationen für diese Kategorie sind z.B. die Komptabilität mit proprietären Dateiformaten, Schnittstellen oder Protokollen (obwohl vom Hersteller keine Komptabilität mit Software von Drittanbieter gewünscht ist). Ein weiteres interessantes Szenario für diese Kategorie ist außerdem der Fall, dass der Original-Quelltext vollständig zerstört wurde, aber noch das kompilierte Programm existiert.

1.3 Anforderungen an einen Obfuscator

Ein Obfuscator nimmt ein beliebiges Programm (in unserem Fall Quelltext) und transformiert es, sodass das Resultat die folgenden Eigenschaften hat [3]:

1. Semantisch äquivalent zu dem Eingabeprogramm **(functionality property)**
2. Höchstens polynomial größer und langsamer **(slowdown property)**
3. Schwer zu untersuchen und zu deobfuscaten **(virtual black-box property)**

Die ersten beiden Eigenschaften lassen sich analytisch für einen gegeben Obfuscator beweisen. Das Vorgehen dabei ist ähnlich zu Korrektheitsbeweisen von Compileroptimierungen, da diese ebenfalls Codetransformationen vornehmen. Die dritte (Blackbox-Eigenschaft) dagegen ist sehr allgemein und problematisch zu untersuchen. Je nach Möglichkeiten, Arten und Ressourcen eines Angreifers, kann die Effektivität eines Obfuscators unterschiedlich ausfallen.

Unterscheidung von Angriffen

2.1 Klassifizierung von Angriffen

Das Vorgehen beim Reverse Engineering einer Software kann grob in verschiedene Kategorien eingeteilt werden. Es lassen sich folgende Arten unterscheiden:

Angriffskategorien	Statische Analyse	Dynamische Analyse
Automatisch		
Menschliche Interaktion		

Abbildung 2.1: Klassifizierung von Reverse Engineering Angriffen

Statisch
Das Programm wird analysiert, ohne es wirklich auszuführen. Beispielsweise das Erstellen eines Kontrollflussgraphen nur durch Untersuchen der Sprungbefehle wäre ein statischer Angriff. Die statischen Methoden können allerdings keine Techniken die zur Laufzeit stattfinden analysieren. Beispielsweise selbst-modifizierender oder zur Laufzeit kompilierender Code kann nicht statisch untersucht werden. Das Programm nicht auszuführen hat im wesentlichen den Vorteil, dass keine Seiteneffekte auf die Umgebung auftreten, wie z.B. durch Schadsoftware, Netzwerkkommunikation oder Modifikation von Konfigurationsdaten.

Dynamisch
Das Programm wird analysiert, während es ausgeführt wird. Ein einfaches Beispiel ist das Beobachten von Speicherinhalten mit einem Debugger. In der Praxis wird in der Regel immer eine dynamische Analyse notwendig sein, da statisch nicht alle Eigenschaften eines Programms untersucht werden können. Eine dynamische Analyse kann allerdings auch durch bestimmte Faktoren, wie beispielsweise nichtdeterministisches Verhalten oder Anti-Debug Mechanismen erschwert werden.

Menschliche Interaktion
Ein Angriff mit menschlicher Interaktion wird definiert als ein Angriff, der nicht vollautomatisch durchgeführt werden kann [2]. Bereits wenn ein Angriff nur wenige Benutzereingaben benötigt, erfordert er damit eine menschliche Interaktion. Wenn Menschen benötigt werden, erhöht sich der Aufwand für einen Angriff und damit auch die Kosten. Es ist schwierig zu bewerten, wie gut ein Programm gegen Angriffe mit menschlicher Interaktion standhalten kann. Die Dauer und der Erfolg eines Angriffs sind dann nämlich wesentlich von den durchführenden Menschen abhänig. Bei riesigen Programmen oder vielen Varianten eines einzigen Programms werden menschlich basierte Angriffe praktisch nicht mehr durchführbar. Ein Beispiel hierfür ist die Anzahl an potenziell neuer Malware pro Tag, diese liegt im Millionenbereich [5, 8].

2.2 Probleme der Source Code Obfuscation

Eine Obfuscation soll in diesem Fall den Quelltext möglichst kompiliziert und schwer nachvollziehbar machen. Zur Entwicklung von Software ist allerdings genau das Gegenteil wichtig, d.h. sinnvolle Variablennamen, klare Strukturen und gute Debugmöglichkeiten. Eine manuelle Obfuscation endet damit sehr schnell im Chaos und es ist unwahrscheinlich, dass diese verschiedensten Angriffen standhält. Aus diesen Gründen ist nur eine automatische Transformation des Quelltexts praktibel. Zur Entwicklung und Debugzwecken kann dann das Programm ganz normal entwickelt werden. Bei einem Release wird dann die Obfuscation durch ein automatisches Tool hinzugefügt. Durch diesen Vorgehen, bzw. durch die Source Code Obfuscation gibt es folgende Nachteile bzw. Probleme:

- Auf Compileroptimierungen muss verzichtet werden
- Performance und Größe des Programms verschlechtern sich in der Regel immer
- Ein Obfuscator könnte unbemerkt Schadcode einfügen
- Der transformierte Quelltext ist bei großen Programmen nicht zu verstehen
- Debugging ist mit dem transformierten Quelltext nicht praktikabel
- Oft ist unklar, wie gut einzelne Transformationen den verschiedensten Angriffsarten standhalten, ohne tiefer in die Implementierungsdetails des Obfuscators einzusteigen

Automatisierte Angriffe

3.1 Allgemeines zu automatisierten Angriffen

Bevor ein Programm ausgeführt werden kann, um dynamisch zur Laufzeit untersucht zu werden, muss sinnvollerweise eine gültige Eingabe existieren. Spätestens wenn keine weiteren Informationen aus dieser einen Eingabe gewonnen können, werden weitere gültige Eingaben benötigt. Für einen vollautomatischen Angriff müssen zunächst einige dieser Eingaben automatisch bestimmt werden. Eine Möglichkeit diese Eingaben zu finden, ist Zufallseingaben zu verwenden. Dieses Vorgehen eignet sich allerdings eher um ungültige Eingaben zu finden ("Fuzzing"). Besonders wenn nur wenige Eingaben aus einer sehr großen Eingabemenge gültig sind, ist dieses Vorgehen nicht praktikabel.

Es gibt einige Ziele beim Reverse Engineering, die eine sehr hohe Code-Coverage erfordern. Hiermit ist gemeint, dass die gefundenen Eingaben („Testfälle“) die meisten Programmpfade abdecken. Um beispielsweise den Kontrollflussgraphen eines Programms mit Obfuscation in sinnvollerweise Weise zu erstellen, dürfen keine Programmpfade fehlen (100% Code-Coverage). Ein weiteres Beispiel ist das Auffinden von selbstüberprüfenden-Code, der sich selbst an mehreren Stellen gegenseitig überprüft (zyklische Abhängigkeiten zwischen den Integritätscheck) [4, 7]. Ein Angreifer müsste alle Überprüfungspunkte im Code finden, was wiederum nur mit einer sehr hohen Code-Coverage möglich ist.

Die letzten beiden Beispiele machen also klar, dass eine hohe Code-Coverage eine wichtige Voraussetzung für das Erreichen weiterer Reverse Engineering Ziele ist. Des Weiteren wurde in 2.1 gezeigt, dass es oft Szenarien gibt, in denen nur vollautomatische Angriffe sinnvoll sind. Die automatische Erzeugung von Testfällen mit möglichst hoher Code-Coverage, ist damit eine sehr wichtige Voraussetzung für die Durchführung von Angriffen, die auf einer dynamischer Analyse basieren.

3.2 Automatische Erzeugung von Testfällen

Die wesentlichen automatischen Methoden zur Testfallerzeugung sind symbolic/concolic exection, model-based testing, combinatorial testing, fuzzing, (adaptive) random testing und searchbased testing [1]. Diese Techniken lassen sich noch in White-Box und Black-Box Testing unterteilen. Das Black-Box Testing erstellt Tests, ohne die interne Struktur des Programms zu berücksichtigen, White-Box Testing dagegen verwendet die interne Struktur, um daraus bessere Tests abzuleiten. Obfuscation verändert nach Definition 1.3 allerdings nicht das Eingabe-Ausgabe-Verhalten, deswegen wird nur White-Box Testing erschwert (abgesehen von kleinen Nebeneffekten wie Performance beim Black-Box Testing).

In dieser Ausarbeitung geht es dabei um die symbolische Ausführung als Angriffsmethode. Die Effektivität einer Transformation wird dabei durch die Verlangsamung der symbolischen Ausführung bewertet. Eine langsamere symbolische Ausführung bedeutet, dass die Transformation resistenter gegen symbolische Ausführung ist.

Überblick Obfuscation/Analysetools

4.1 Obfuscation- und Analysetools

Dieser Bereich soll einen groben Überblick über die Tools in der Praxis geben, die zur Obfuscation oder zum Reverse Engineering eingesetzt werden können. Nicht aufgeführt sind dabei öffentlich nicht verfügbare Tools wie z.B. Microsoft SAGE.

Tools für symbolische Ausführung

Tool:	Sprachen:	Url:
KLEE	LLVM	`https://klee.github.io/`
JPF	Java	`http://babelfish.arc.nasa.gov/trac/jpf`
jCUTE	Java	`https://github.com/osl/jcute`
janala2	Java	`https://github.com/ksen007/janala2`
KeY	Java	`http://www.key-project.org/`
S2E	x86, x86-64, ARM	`http://s2e.epfl.ch`
Otter	C	`https://bitbucket.org/khooyp/otter/overview`
Pex	.NET Framework	`http://research.microsoft.com/en-us/projects/pex/`
Jalangi2	JavaScript	`https://github.com/Samsung/jalangi2`
pysymemu	x86-64	`https://github.com/feliam/pysymemu/`
Triton	x86/x86-64	`http://triton.quarkslab.com`
angr	x86, x86-64, ARM, ..	`http://angr.io/`

Tabelle 4.1: Analysetools mit symbolischer Ausführung

Tools für Obfuscation

Tool:	Sprachen:	Url:
tigress	C	`http://tigress.cs.arizona.edu/`
ProGuard	Java	`https://www.guardsquare.com/en/proguard`
Dotfuscator	.NET Framework	`https://www.preemptive.com/products/dotfuscator`
ConfuserEx	.NET Framework	`https://yck1509.github.io/ConfuserEx/`
Stunnix	C/C++	`http://stunnix.com/prod/cxxo/`
Obfuscator-LLVM	LLVM	`https://github.com/obfuscator-llvm`

Tabelle 4.2: Tools für Obfuscation

Transformationen für Obfuscation

Im Folgenden werden ein paar Quelltext-Transformationen mit tigress vorgestellt. Diese Transformationen lassen sich beliebig kombinieren und hintereinanderschalten. Tigress selbst kann immer nur eine Quelltextdatei verarbeiten, kann jedoch auch mehrere Dateien zu einer zusammenfassen (für größere Projekte). Sofern die Transformation nicht zu kompliziert ist, wird noch ein kurzes Quelltextbeispiel angegeben. Diese Beispiele sind teilweise etwas vereinfacht und deutlich lesbarer formatiert, als der ursprünglich transformierte Quelltext. Die Beispiele sollen nur die Prinzipen verdeutlichen.

5.1 Encode Literals

Literale (in tigress nur integers und strings) werden erst zur Laufzeit durch eine Funktion erzeugt und nicht statisch im Programm gespeichert. Dieses Verfahren schützt effektiv nur vor statischen Analysen. Zur Laufzeit liegen die Konstanten (spätestens wenn sie gebraucht werden) ganz normal im Speicher.

```c
int main(void) {
    char* magic = "42";
    printf("%s", magic);
    return (0);
}
```

obfuscate

```c
char litStr0[3];
char litStr1[3];

int main(void)
{
    char *magic;
    /* ... */

    stringEncoder(0, litStr0);
    magic = (char*)(litStr0);

    stringEncoder(1, litStr1);
    printf((char const)(litStr1), magic);
    return(0);
}
```

```c
void stringEncoder(int n , char str[])
{
    int x = 0;
    switch (n) {
        case 1:
            str[x] = '%';
            x++;
            str[x] = 's';
            x++;
            str[x] = '\000';
            x++;
        break;
        case 0:
            str[x] = '4';
            x++;
            str[x] = '2';
            x++;
            str[x] = '\000';
            x++;
        break;
    }
}
```

Abbildung 5.1: Beispiel EncodeLiterals

5.2 Encode Arithmetic

Arithmetische Ausdrücke (Integer) werden in komplexere äquivalente Ausdrücke umgewandelt. Für Integer Ausdrücke können auch binäre Operationen verwendet werden. Wenn es sich um die Berechnung einer Konstante handelt, ist der Effekt allerdings sehr gering, da der Wert zur Laufzeit (wenn berechnet) einfach im Speicher liegt.

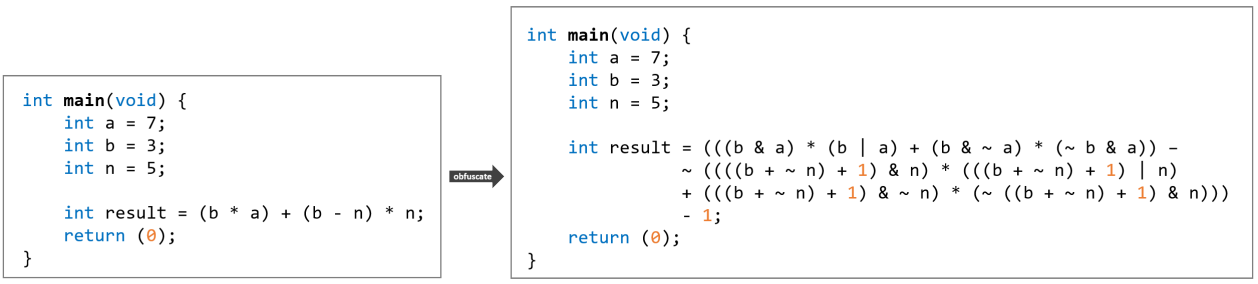


Abbildung 5.2: Beispiel Encode Arithmetic

5.3 Control flow flattening

Der Kontrollfluss wird in eine flache Hierarchie transformiert, in der alle Codeblöcke gleiche Vorgänger und Nachfolger haben. Die Struktur des Kontrollflusses ist ein wichtiger Bestandteil eines Programms und kann durch diese Transformation recht gut verschleiert werden. Außerdem wird das Erstellen eines (sinnvollen) Kontrollflussgraphen schwieriger. Optional können hier auch weitere Konstrukture wie gotos oder dead-code eingefügt werden, um die Struktur noch stärker zu verschleiern.

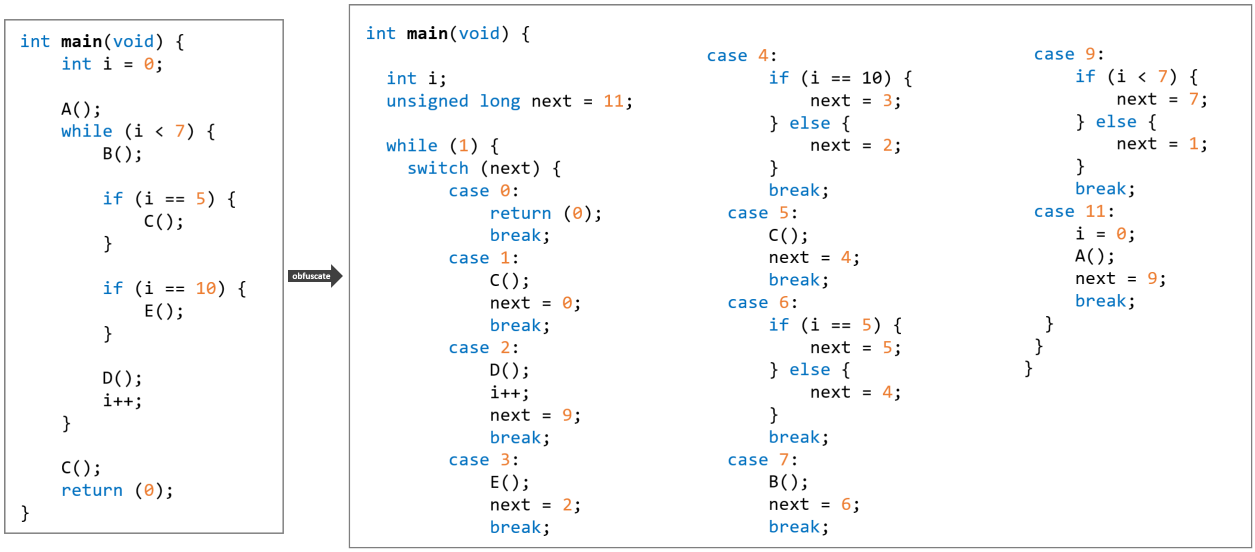


Abbildung 5.3: Beispiel Control flow flattening

5.4 Opaque predicates

Erzeugt künstliche Verzweigungen, die zur Laufzeit immer oder niemals ausgeführt werden. Die Bedingungen dieser Verzweigungen sind also effektiv Konstanten. Besonders geeignet sind Bedingungen, die sich statisch nur schwer untersuchen lassen wie z.B. rekursive Datenstrukturen. Optional können die Variablen zur Auswertung dieser Bedingungen neue Werte erhalten, aber dennoch den vorherigen konstanten Wert beibehalten (UpdateOpaque).

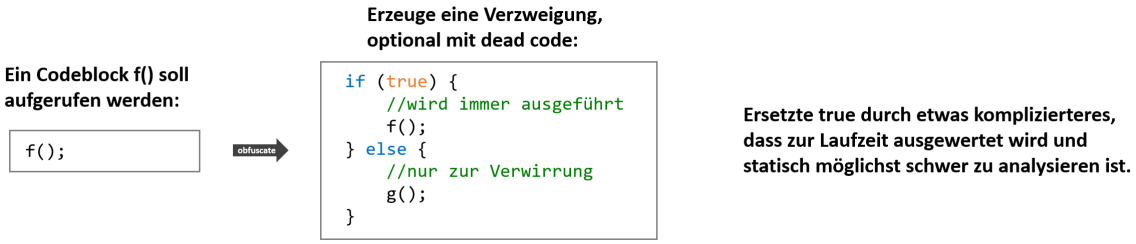


Abbildung 5.4: Prinzip Opaque predicates

5.5 Merge Functions

Mehrere Funktionen werden zu einer Funktion zusammengefasst. Die aufzurufende Funktion wird dann über einen Parameter identifiziert. Für einige Programmiersprachen kann es problematisch sein, wenn die zusammengefassten Funktionen verschiedene Rückgabetypen haben. Diese Problematik besteht für C nicht, da typlose Zeiger möglich sind. Optional kann der Kontrollfluss in den zusammengefassten Funktionen wieder geglättet werden (MergeFlatten).

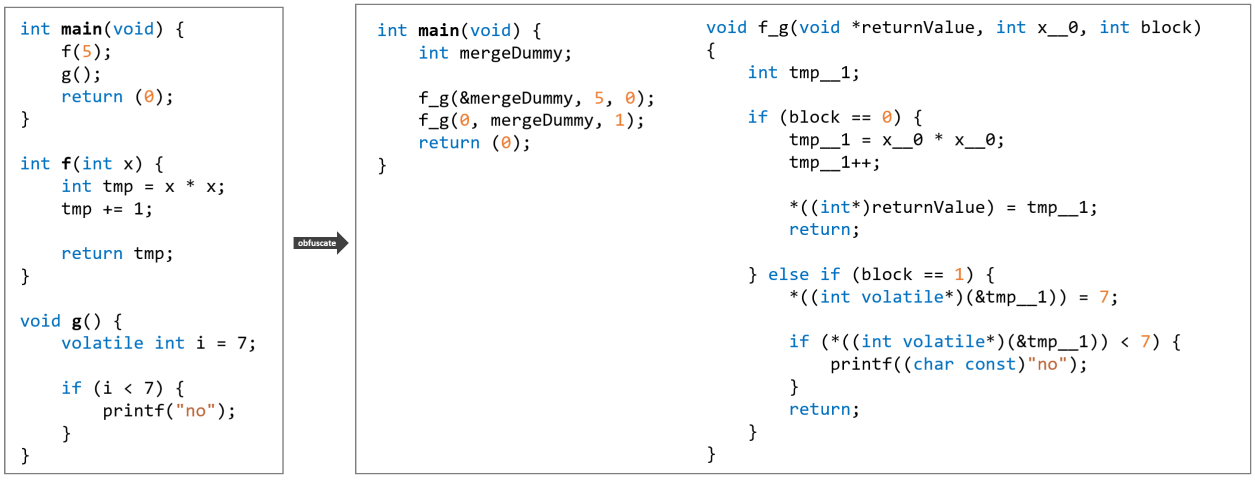


Abbildung 5.5: Beispiel Merge Functions

5.6 Split Function

Eine Funktion wird in mehrere Teilfunktionen aufgeteilt. Dies erfordert allerdings, dass eine Funktion sich auch aufteilen lässt, d.h. voneinander unabhängige Blöcke innerhalb der Funktion existieren. Durch diese Transformation wird der Kontrollflussgraph größer und damit unübersichtlicher. Durch extrem viele Funktionsaufrufe kann sich allerdings auch die Performance wesentlich verschlechtern.

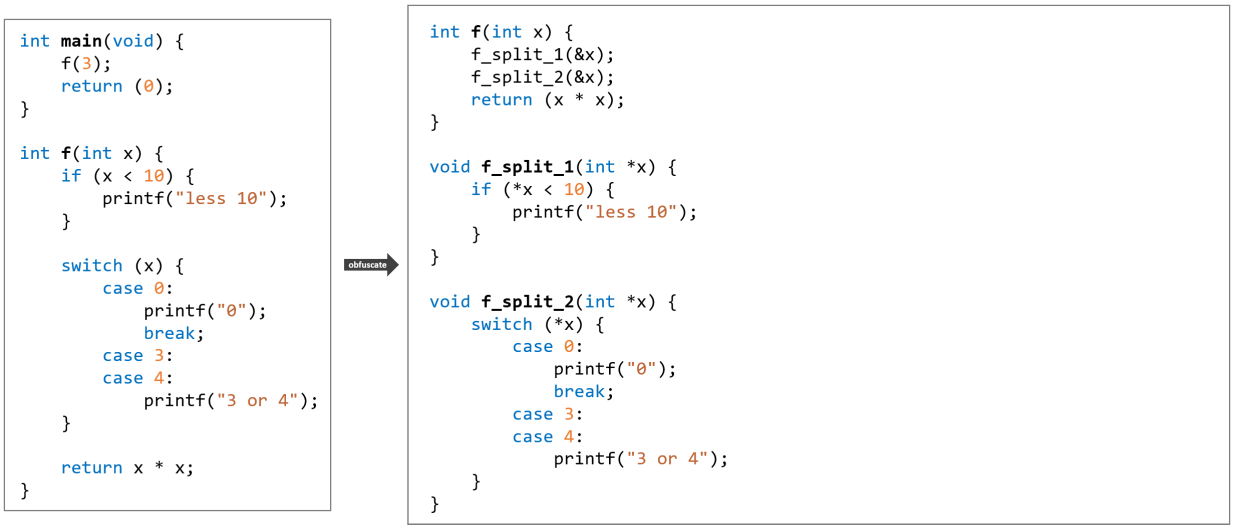


Abbildung 5.6: Beispiel Split Function

5.7 Virtualize

Wandelt eine Funktion in einen Interpreter um, dessen Bytecode-Sprache für die ursprüngliche Funktion spezialisiert ist. Für Seiteneffekte bzw. Funktionsaufrufe werden zusätzliche Befehle in der Bytecode-Sprache reserviert, um eine Interaktion des Interpreters mit dem normalen Programm zu ermöglichen. Bei der Transformation wird ein Zufallsgenerator eingesetzt, um die Bytecode-Sprache und die Instruktionen zu variieren. Die Ausführung auf dieser virtuellen Hardware ist sehr langsam und eignet sich daher nicht für Performance kritische Anwendungen.

5.8 Jitting

Eine Funktion wird durch eine andere Funktion erst dynamisch zur Laufzeit kompiliert. Verwandt sind Verfahren, wie Entpacken oder Entschlüsseln von Instruktionen. Der Vorteil bei dieser Transformation ist, dass eine Funktion erst erzeugt wird, wenn sie auch tatsächlich benötigt wird. Bis zu diesem Zeitpunkt kann sie also praktisch nicht analysiert werden.

5.9 Dynamic Jitting

Dies funktioniert analog zum Jitting, der Unterschied besteht darin, dass der Code zum Kompilieren zur Laufzeit geändert wird. Dabei wird sichergestellt, dass der zu erzeugende Code semantisch äquivalent bleibt.

Effektivität von Obfuscation

6.1 Probleme der Messbarkeit

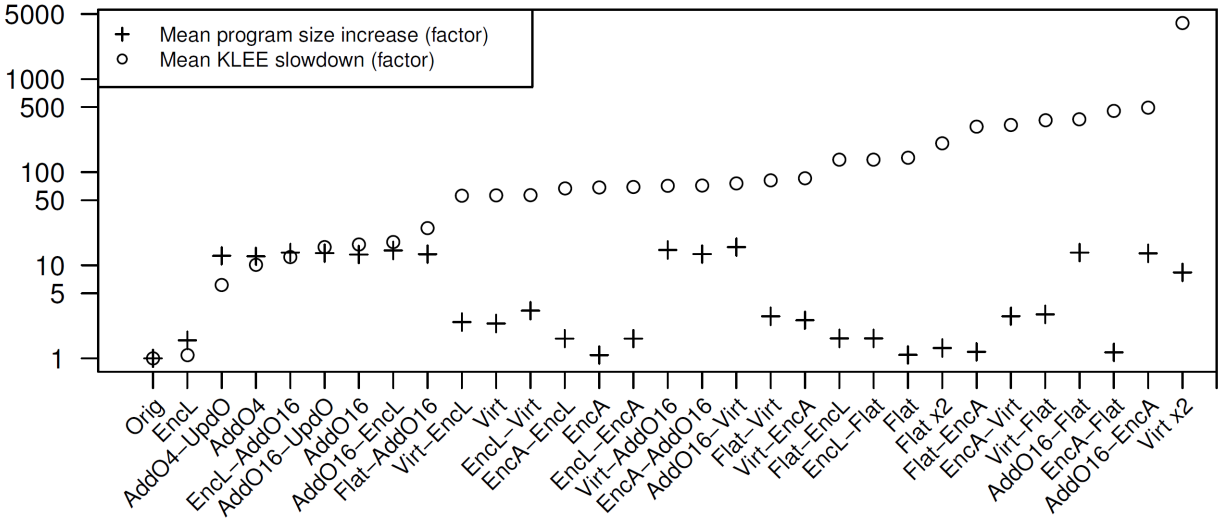
Das Prinzip der Obfuscation liegt grob gesagt darin, einen Angriff durch verwirrende Strukturen zu erschweren. Vollständig verhindert werden kann ein Angriff dadurch aber nicht. Der Erfolg bzw. die Effektivität einer Obfuscation ist daher schlecht messbar, besonders wenn ein Angriff von unregelmäßigen Faktoren (wie z.B. menschliche Interaktion) abhängig ist.

Als Maß für den Erfolg einer Transformation wird hier die Verlangsamung der symbolischen Ausführung verwendet. Natürlich gibt es auch andere Angriffsarten, wie z.B. die statischen Verfahren, diese werden dadurch natürlich nicht berücksichtigt. In [2] wird dieser Verlangsamungsfaktor wie folgt berechnet: KLEE wird verwendet um Tests mit 100% Code-Coverage zu erzeugen. Die Zeit des transformierten Programms wird dann durch die Zeit für das nicht transformierte Programm geteilt. Um dabei kleine Schwankungen (z.B. OS-Hintergrundprozesse) auszugleichen, wird die Ausführung 10-mal wiederholt und der Durchschnittswert berechnet. Ein höherer Faktor weist dann auf eine höhere Resistenz gegen symbolische Ausführung hin.

Ein Problem bei dieser Vorgehensweise ist, dass nicht nur die Verlangsamung der symbolischen Ausführung gemessen wird, sondern auch die der Ausführung selbst. Transformationen die also die Performance des Programms verschlechtern, werden automatisch etwas besser bewertet.

6.2 Verlangsamung symbolischer Ausführung

Encode Literals (**EncL**), Opaque predicates (**AddO**), UpdateOpaque (**UpdO**), Control flow flattening (**Flat**), Virtualize (**Virt**), Encode Arithmetic (**EncA**).



Quelle: Figure 1: Impact of obfuscation on the KLEE symbolic exection for programs in 1st dataset. [2]

Abbildung 6.1: Auswirkungen von Obfuscation auf symbolische Ausführung

Symbolische Angriffe

7.1 Klassische symbolische Ausführung

Symbolische Ausführung (symbolic exection oder symbolic evaluation) ist ein Verfahren zur Programmanalyse, bei dem Eingaben eines Programms bestimmt werden sollen, unter denen das Programm bestimmte Teile (Pfade) ausführt. Im Gegensatz zur normalen (”konkreten”) Ausführung eines Programms, werden einige Variablen mit symbolischen Werten und nicht mit konkreten Werten belegt. Diese Variablen sind in der Regel sinnvollerweise Eingabevariablen des Programms. Das Programm wird Schritt für Schritt von einem Interpreter (Abstrakte Interpretation) abgearbeitet, dieser wird als Symbolic Execution Engine bezeichnet. Für Variablen die abhängig von den symbolischen Variablen sind und Veränderungen an symbolischen Variablen, werden durch Ausdrücke repräsentiert. Diese Ausdrücke enthalten dadurch nur symbolische Variablen als unbestimmte Größen. Wenn die symbolische Ausführung auf eine Verzweigung trifft, wird die Simulation des Programms aufgespaltet. Somit werden beide Fälle (true und false) berücksichtigt. Es entsteht eine Baumstruktur, die alle möglichen Pfade des Programms repräsentiert.

```
int f(int x, int y) {
    int z = 2 * x;

    if (x == 100) {
        z = z - y;
    }

    z = z + y;

    if (y < z) {
        assert(0);
    }
}
```

f(100, 0) -> Assertion schlägt fehl
f(100, 200) -> Alles OK

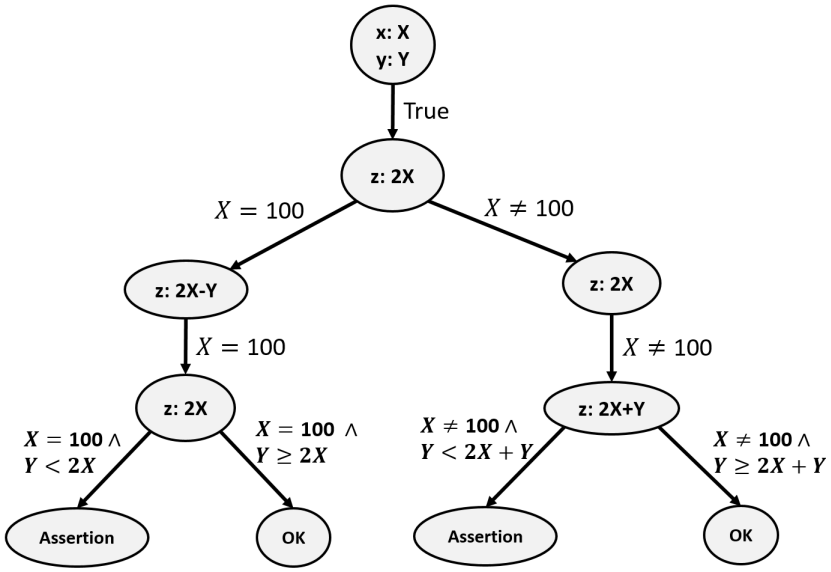


Abbildung 7.1: Beispielbaum für eine symbolische Ausführung

Auf dem Weg zu einem Blatt des Baums sammeln sich mehrere Bedingungen an, die alle erfüllt werden müssen, damit das Programm genau diesen Weg nimmt. Wenn alle diese Bedingungen logisch UND-verknüpft werden, erhält man eine boolesche Formel. Alle gültigen Belegungen dieser Formel sind Eingaben, unter denen das Programm den entsprechenden Pfad ausführt. Ob eine Formel überhaupt lösbar ist und das Lösen der Formel, werden von einem separaten Programm durchgeführt, dieses Programm wird SMT-Solver genannt. Das Problem eine solche Formel zu lösen wird als SAT (Boolean satisfiability problem) bezeichnet.

7.2 Probleme von symbolischer Ausführung

Eine Problematik ist, dass Schleifen und Rekursionen einen unendlich großen Programm-Pfad-Baum erzeugen. Dies ist prinzipiell auch sinnvoll, da Schleifen auch Endlosschleifen sein können. Ein weiteres Problem ist die Interaktion mit der Umgebung, wie z.B. Variablen auf dem Heap, Konfigurationsdateien, Zugriff auf Datenbanken oder Netzwerkkommunikation. Ein weiteres Problem sind Systemcalls, weil die Symbolic Execution Engine während eines solchen Systemaufrufs keine Kontrolle mehr über das Programm hat.

Das Lösen einer boolschen Gleichung, um schlussendlich eine Eingabe für einen bestimmen Pfad zu finden, ist ebenfalls recht schwierig. Das SAT (Boolean satisfiability problem) ist im allgemeinen Fall NP-vollständig. Damit SAT-Solver eine Lösung in einer praktikablen Zeit finden können, werden daher approximative Verfahren genutzt, die nicht immer eine Lösung finden, obwohl vielleicht eine Lösung existiert. Eine Weiterentwicklung bzw. Verbesserung der SAT-Solver hilft somit auch der symbolischen Ausführung.

7.3 Concolic Execution

Unter Concolic Exection (auch Concolic testing oder Symbolic and Concrete Execution) ist eine Software Test- bzw. Verifikationstechnik, die eine symbolische und konkrete Ausführung nutzt. Das Ziel ist auch hier automatisch Testfälle mit einer möglichst hohen Code-Coverage zu erzeugen. Der Vorteil gegenüber der einfachen, klassischen symbolischen Ausführung liegt darin, dass auch komplexere Programme aus der Praxis ausgeführt werden können. Die meisten Probleme aus 7.2 können somit umgangen werden. Allerdings muss auch hier noch ein SAT-Solver eine boolesche Gleichung lösen. Ursprünglich beschrieben wurde diese Technik als DART (Directed Automated Random Testing) [6].

Es werden analog zur symbolischen Ausführung zunächst symbolische Variablen (Eingabevariablen) festgelegt. Es wird dann eine konkrete Ausführung des Programms durchgeführt. Die Eingabevariablen erhalten hierzu beliebige Werte. Während das Programm normal ausgeführt wird, werden Operationen auf symbolischen Variablen (die zu diesem Zeitpunkt noch konkrete Werte besitzen), sowie Verzweigungen in einem Tracefile geloggt. Sobald das Programm terminiert ist, wird die eigentliche symbolische Ausführung gestartet, diese nutzt jetzt allerdings nicht mehr das ursprüngliche Programm sondern das durch die normale Ausführung erzeugte Tracefile. Es wird dann natürlich nicht der komplette Programm-Pfad-Baum erzeugt, sondern nur ein Ausschnitt der die vorherige konkrete Ausführung repräsentiert. Die letzte Bedingung in dem Baum (die noch nicht zuvor negiert wurde) wird negiert. Um eine Eingabe zu erzeugen die dann unter dem Programm diesem neuen Pfad (letzte Bedingung negiert) folgt, wird dann ein SAT-Solver eingesetzt. Wenn keine Bedingungen zum Negieren mehr übrig sind wird das Programm erneut konkret ausgeführt und es wird versucht weitere Programmpfade zu finden.

Anti-Symbolic Execution Obfuscation

Die bisherigen Obfuscation Transformationen können symbolische Angriffe nur verlangsamen. Ein Grund dafür ist, dass diese Transformationen im wesentlichen Komplexität zu internen Programmteilen hinzufügen, allerdings keine Komplexität die auf Eingabevariablen (symbolischen Variablen) beruht. Im Folgenden werden ein paar Transformationen aus [2] vorgestellt, die speziell gegen symbolische Ausführung resistent sind.

8.1 Path explosion / Path divergence

Die Concolic Execution erkundet zunächst immer nur einen Pfad. Wenn es extrem viele Pfade gibt, wird dieses Vorgehen nicht mehr praktikabel. Auch die klassische symbolische Ausführung hat Probleme mit riesigen Programm-Pfad-Bäumen. Die Idee einer Path explosion ist es, diese Situation künstlich herzustellen, um die symbolische Ausführung mit Pfaden zu überhäufen. Beispielsweise erzeugt eine einfache Schleife, deren Abbruchbedingung eine symbolische Variable enthält, eine Path explosion. Bei jedem Schleifendurchlauf wird dann die Symbolic Execution Engine einen weiteren Pfad anlegen.

```c
int main(int argc, char** argv) {
    char *p = argv[1];
    int containsA = 0, containsB = 0, containsC = 0;

    while (*p != '\0') {
        switch (*p) {
            case 'A': containsA = 1; break;
            case 'B': containsB = 1; break;
            case 'C': containsC = 1; break;
        }

        p++;
    }

    return (0);
}
```

Abbildung 8.1: Beispiel für eine Path Explosion

Bei der Path divergence werden symbolische Variablen für unbedingte Sprünge benutzt. Dies funktioniert ähnlich wie die Flatten Transformation (5.3). Durch diesen impliziten Kontrollfluss werden die Vergleiche bzw. Abhänigkeiten zwischen symbolischen Variablen und Konstanten entfernt. Erzeugte Tests haben dann nichts mehr mit dem eigentlichen Programmpfad zu tun.

8.2 Range Dividers

Range Dividiers können an beliebigen Positionen im Code eingefügt werden. Sie teilen den Eingabebereich im mehrere Pfade auf, die alle semantisch äquivalenten Code enthalten. Die Symbolic Execution Engine wird dann alle Pfade erkunden, aber keine neuen Informationen erhalten. Neben der höheren Anzahl an Pfaden bietet dieses Prinzip einen weiteren Vorteil: Einzelne Pfade können verschiedene Transformationen enthalten. Die Symbolic Execution Engine wird dann Eingaben erzeugen, die zwar verschiedene Programmpfade verfolgen, aber dennoch semantisch äquivalentes Verhalten im Programm auslösen.

```
int main(int argc, char** argv) {
    unsigned char *str = argv[1];
    unsigned int hash = 0;

    for (int i = 0; i < strlen(str); i++, str++) {
        char c = str[i];

        switch (c) {
            case 1:
                hash = hash * 31 - c;
                break;
            case 2: //obfuscated version of case 1
                break;
            case 3: //another obfuscated version of case 1
                break;
            //...
        }
    }

    return (0);
}
```

Abbildung 8.2: Beispiel für einen Range Divider

8.3 Invariant inputs

Ein Problem der Range Dividers ist, dass sie nur signifikante Auswirkungen haben, wenn sie in Schleifen verwendet werden. Generell ist ein sehr simpler Code schwierig zu obfuscaten. Diese Transformation kann gut für sehr simplen Code eingesetzt werden. Allerdings wird dazu das Eingabe-Ausgabe-Verhalten des Programms verändert, was gegen die Functionality property aus 1.3 verstößt. Nur noch festgelegte Eingaben des Benutzers erzeugen dann das normale Programmverhalten, bei anderen Eingaben ist das Verhalten undefiniert. Natürlich muss es für die entsprechende Anwendungen akzeptabel sein, für ungültige Eingaben ein undefiniertes Verhalten zu besitzen.

Beispielsweise kann die Virtualize Transformation (5.7) dazu verwendet werden. Mit der Benutzereingabe wird dann der Bytecode dekodiert. Nur bei der richtigen Eingabe wird dann das ursprünglich gewünschte Programm ausgeführt. Die Effektivität dieser Transformation ist die höchste aus [2].

Zusammenfassung

Die Transformationen aus Kapitel 5 können eine symbolische Ausführung nur verlangsamen. Auch wenn die Statisik aus [2] nur mit kleinen Programmen getestet wurde, lässt sich die Effektivität einzelner Transformationen grob einordnen. Bei größeren Programmen kann die symbolische Ausführung schneller an ihre Grenzen stoßen. Allerdings kann während des Reverse Engineerings manchmal ein interessanter Programmteil identifiziert werden, der dann genauer untersucht wird. Beispielsweise könnte ein Programm aus Performance-Gründen nur sicherheitskritische Teile obfuscaten.

Eine Problematik bei der Source Code-Obfuscation ist, dass oft unklar bleibt, wie gut einzelne Transformationen gegen welche Angriffsarten eingesetzt werden können. Eine exakte analytische Aufarbeitung ist für die meisten Transformationen nicht möglich und empirische Untersuchungen geben nur einen Einblick in die derzeitigen Möglichkeiten. Verbesserungen von symbolischer Ausführung, SAT-Solvern oder anderen Methoden, könnten in Zukunft jederzeit die Effektivität von Obfuscation verschlechtern. Damit ist Obfuscation weit entfernt von der Zuverlässigkeit von kryptographischen Verfahren. Obfuscation hat insgesamt eher einen kreativen Charakter (Art der Transformation, Reihenfolge, künstlicher Deadcode, etc.) und kann Angriffe oftmals nur verlangsamen. Trotzdem wird Obfuscation in der Praxis verwendet, für z.B. das Schützen von proprietären Daten oder Verschleiern von Schadsoftware. Damit bleibt Obfuscation weiterhin praxisrelevant.

Literaturverzeichnis

[1] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001, August 2013.

[2] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. pages 189–200, 12 2016.

[3] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6:1–6:48, May 2012.

[4] Hoi Chang and Mikhail J. Atallah. *Protecting Software Code by Guards*, pages 160–175. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[5] Symantec Corporation. Internet security threat report 2016. Technical Report April 2016. `https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf`.

[6] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[7] Bill Horne, Lesley Matheson, Casey Sheehan, and Robert E. Tarjan. *Dynamic Self-Checking Techniques for Improved Tamper Resistance*, pages 141–159. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[8] McAfee. Mcafee labs threats report. Technical Report March 2016. `http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-mar-2016.pdf`.

[9] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv.*, 49(1):4:1–4:37, April 2016.