FernUniversität in Hagen — Fakultät für Mathematik und Informatik

Obfuskation und Malware Evasion durch Verlagerung von Funktionalität in Threads

Oliver Deja
Matrikelnr.: 6695710

Abschlussarbeit im Studiengang
Master of Science Praktische Informatik

eingereicht im August 2021

Einleitung

- Engineers vs. Reverse Engineers
- Obfuskation
- Kontrollflussgraph
- Beiträge:
  - Implementierung
  - Untersuchung
  - Vergleich

- Schlagworte zu sehen
- Kampf zw. Software Ingenieure, die Software entwickeln, und Reverse Engineer, welche versuchen dieses Produkt für verschiedene Zwecke zu analysieren
- Obfuskation als Gegenmaßnahme gegen Reverse Engineering
- Ein Ansatz ist die Verschleierung des Kontrollflussgraphen, so auch hier
- Im Folgenden wird näher auf diese Begriffe eingegangen

- Beitrag besteht aus 3 Teilen
- konkrete Implementierung vorgestellt
- diese hinsichtlich verschiedener Gesichtspunkte untersucht
- abschließend in wissenschaftl. Kontext verglichen

Inhalt

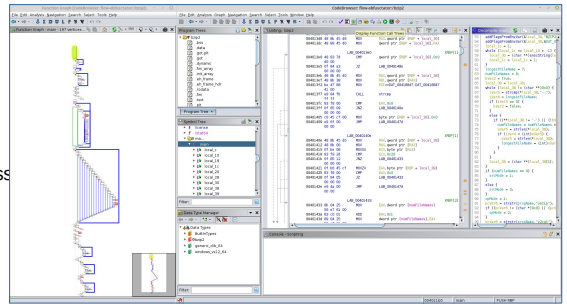
- Zuerst Grundlagen
- dann die für die Masterarbeit erstellte mögliche Implementierung vorgestellt
- anschließend diese Implementierung in Evaluation untersucht
- abschließend die Erkenntnisse aus den vorangegangenen Teilen weiterführend betrachtet
- Zuletzt die relevantesten Aspekte der Arbeit zusammengefasst und einen Ausblick gewährt

Inhalt

Allgemein die Grundlagen kurz umrissen, die mit dem vorgestellten Thema zusammenhängen und auf welchen die Idee hinter dieser speziellen Obfuskation basiert

Grundlagen

- Kontrollflussgraph
  - Jump, Call, Return
- Reverse Engineering
- Obfuskation
  - Gestaltung, Daten, Kontrollfluss
- Malware Detektion
  - Signatur, Verhalten, Heuristik
- Malware Evasion



Ghidras Benutzeroberfläche

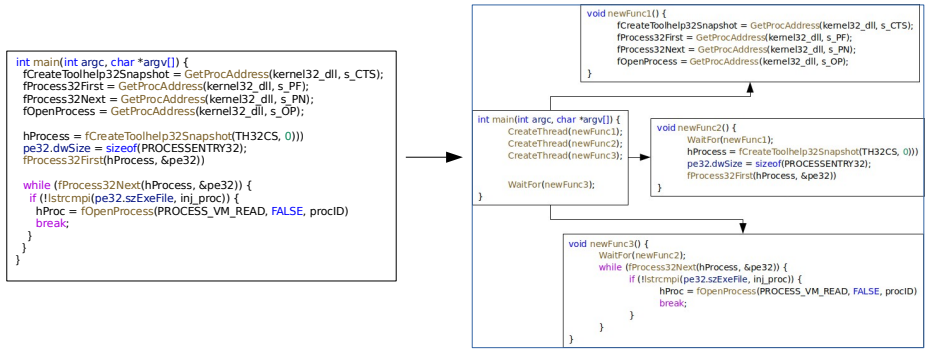
Inhalt

- CFG: weil wesentlicher Bestandteil; Programmablauf; seriell, wenn nicht Instr. Einfluss …; Call/Ret nicht im Graph, Jmp Kante
- Wiedergewinnung der konkreten Implementierung bei Produkten, zu welchen keine Informationen über ihre innere Beschaffenheit vorliegt
  verschiedene Codeformen, teilweise automatisiert möglich
  Disassemblierung, Dekompilierung, automatische Annotationen, graphische Darstellung mit Tools wie Ghidra
  Ziele: Schwachstellen, bösartiges Verhalten, Manipulation, geistiges Eigentum
- erschweren; Funcnames, Crypt, Instr. einfügen
- Hash/LSH/Yara(Muster/Regex)→DB; stat./dyn., sys calls / allocas mit permissions; AI, Kombi → Wahrsch. als Phase 1
- Obfuskation → Polymorphie

Übersicht gezeigt
Entwurfsentscheidungen und Einschränkungen vorgestellt
Abschließend konkrete Implementierung geschildert

nur eine Möglichkeit zur Implementierung, bei der selbstverständliche Teilaspekte durch Alternativen ersetzt werden können, einige werden auch später noch vorgestellt

Übersicht



nicht obfuskiert obfuskiert

Mit dieser Folie soll das Prinzip der Obfuskation skizziert werden

- links nicht obfus. mit vereinfachtem C-Code Bsp.
- unterteilt durch Leerzeile, 3 neue funcs, als Thread gestartet warten jeweils auf den Vorgänger

Hier kann schon die erste Konsequenz hinsichtl. der veränderten Form des CFG antizipiert werden

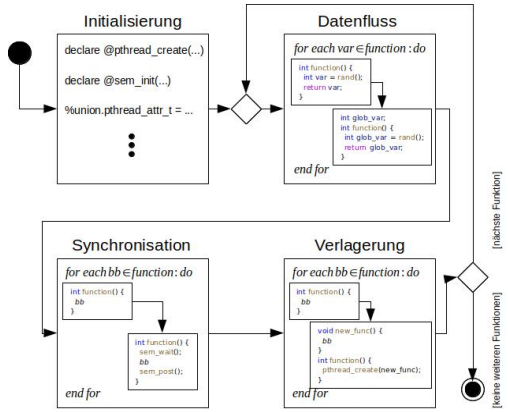
Auch lassen sich erste Implikationen bzgl. Implementierung erkennen: Datenfluss muss geändert werden

Entscheidungen und Einschränkungen

- **Rahmenwerk**: LLVM (IR) Optimierungsdurchlauf
- **Zielbetriebssystem**: Windows und Linux, 32- und 64-bit
- **Quellsprache**: C und C++
- **Parallelisierungsmechanismen**: Systemaufrufe
- **Datenflussanpassungen**: über globale Variablen
- **Einteilungsgröße**: ein Basic Block
- **Besonderheiten**: rekursiv, variable Parameter, Ausnahmebehandlung

- ausgehend von nativer Code, weil weniger gut analysierbar und beste Kontrolle, und möglichst generisch und unkomplex, weil nur abhängig von Rahmenwerk (diverse Langs) → LLVM Pass
  Entscheidung unterstützt durch die Nutzung in anderen wissenschaftl. Publikationen
- Windows, weil starke Verbreitung und begehrtes Ziel; Linux, weil Entwicklung leichter (hat sich bestätigt); Prozessorarchitektur nicht groß unterschiedlich, deshalb beide
- Offizieller Support nur C/C++, aber auch geeignet
- Weil die Alternativen wie Stack/Heap höheren Verwaltungsauffwand
- Weniger komplex, letztlich aber beliebig (Param)
- rekursiv und var Param wegen Datenfluss decision
- durch Thread kreierte Schicht trennt Entstehungsort und Routine, auch Win/Linux

Architektur und Arbeitsweise



Aufgrund umfangreiche Implementierung soll nicht auf Details eingegangen werden

- Systemfunktionen und Systemtypen, wichtig: CreateThread und Semaphor (Create, Wait)
  Notwendig, die Systemobjekte aufzuräumen (Rsrc.)
- „Globalisierung“, LLVM Identifier, Optimierung durch Prüfung, wo Identifier verwendet, PHI-Nodes: Variable abhängig Vorgänger (komplex)
- vor Verlagerung, weil für die Sync notwendige Beziehungen zwischen den einzelnen Funktionalitäten noch vorhanden
  am Anfang Wartepunkt am Ende Freigabepunkt gemäß Verweigungsinstruktion injiziert, Spezial: erste nicht, letzte gibt ursprüngliche Funktion frei
- Verschieben und CreateThread eingefügt, bei mehrfachen Durchlaufen selbst neu starten
- Besonderheit Windows: Terminierung Gefahr, weil nicht klar wann, Mutex nicht freigegeben
  Lösung: Zählervariable, weil am wenigsten komplex und Verwaltungsaufwand

Inhalt

- Beweis der Korrektheit des obfus. Prog. (sonst hat die Obfuskation an sich ja keinen Sinn)
- Formelle Obfuskationsmetriken
  - Wirksamkeit, Widerstandsfähigkeit, Kosten
- Auswirkungen auf das Reverse Engineering
- Auswirkungen auf die Malware Detektion

Korpus: zwei Beispielprogramme, eines zur Berechnung von einer bestimmten Stelle von PI, das andere bzip2 zur Kompression; zudem noch die Coreutils, Standardprogramme des GNU Betriebssystems, beliebt für Evaluation

Malware bestehend aus allen Quellcodes die aufgefunden werden konnten, weil Binary Rewriting keine Alternative

Beweis der Korrektheit des obfuskierten Programms

- Vergleich zwischen obfuskiertem und nicht obfuskiertem Programm
- Coreutils Testsuite

obfuskiert

```
Coreutils
============
# TOTAL: 591
# PASS:  353
# SKIP:  128
# XFAIL: 0
# FAIL:  89
# XPASS: 0
# ERROR: 21
```

```
Gnulib
============
# TOTAL: 320
# PASS:  200
# SKIP:  37
# XFAIL: 0
# FAIL:  83
# XPASS: 0
# ERROR: 0
```

nicht obfuskiert

```
Coreutils
============
# TOTAL: 600
# PASS:  485
# SKIP:  109
# XFAIL: 0
# FAIL:  0
# XPASS: 0
# ERROR: 6
```

```
Gnulib
============
# TOTAL: 322
# PASS:  286
# SKIP:  36
# XFAIL: 0
# FAIL:  0
# XPASS: 0
# ERROR: 0
```


Mit beiden Varianten des bzip2 Programms wurden verschiedene Testdaten komprimiert, ebenfalls wurden verschiedene Kommandozeile angewandt
Es konnte weder bei den Dateiausgaben noch bei den Kommandozeilenausgaben des Programms Unterschiede festgestellt werden.

die Coreutils bieten automatisierte Tests, nicht obfuskiert wie erwartet bei fast 100 %
obfuskiert zwar nur bei fast 75 %, als Stabilität aber genügend, wenn beachtet wird, dass bestimmte Konstrukte inhärent nicht mit der Obfuskation funktionieren, z.B. signal-Funktion oder bestimmte Annahmen über das Programm bei den Tests

Formelle Obfuskationsmetriken

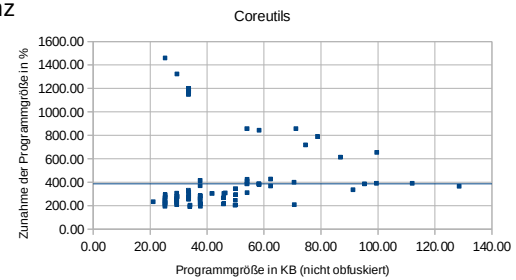
- Coreutils für die Messungen
- zu beachten: verschiedene Abhängigkeiten, wie Einteilungsgröße

Die Qualität einer Obfuskation kann anhand der Wirksamkeit, der Widerstandsfähigkeit und anhand der Sparsamkeit bzw. anhand der Kosten bestimmt werden

Bewusst diese Darstellung gewählt, weil Änderungen eines Aspekts die anderen Aspekte beeinflussen kann. Dementsprechend befinden sich diese Merkmale in einem Spannungsverhältnis.

Formelle Obfuskationsmetriken (Wirksamkeit)

- Zunahme der Programmgröße
- Zunahme von Funktionen
- Längste gemeinsame Teilsequenz
- Zunahme der Instruktionen



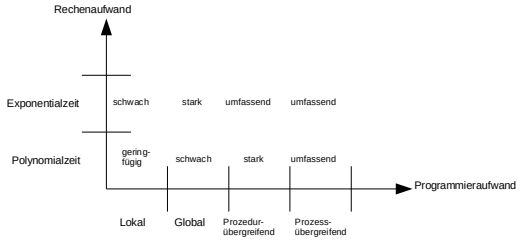
orientiert sich an Software Engineering Metriken, wobei das Qualitätsmaß dem des Software Engineerings entgegengesetzt ist

Zunahme der Programmgröße
- zwischen 190 – 1500 %, bei höherer Ausgangsprogrammgröße Trend zum Mittelwert
- es ist davon auszugehen, dass Spitzenwerte aufgrund hoher Dichte spezieller Eigenschaften
- erst im Zusammenhang mit Kosten und Widerstand, weil sonst unendlich

- Zunahme von Funktionen stark abhängig Einteilungsgröße, entspricht Anzahl Bbs
- weitestgehend unauffällig bei Vergleich mit beliebigen Programmen
- identisch mit Programmgröße
- sonst keine Änderungen hinsichtlich OOP (SE), aber ggf. gefühlte Wirksamkeit durch Distanz

Formelle Obfuskationsmetriken (Widerstandsfähigkeit)

- Aufwand zur Erstellung eines Deobfuskators



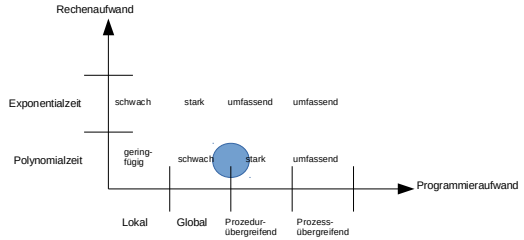
das Maß reicht von geringfügig bis irreversibel, irreversibel nur bei der Frage, ob Variable vorher lokal oder global, für deob. aber irrelevant
das Maß wird zweidimensional anhand Rechenaufwand und Programmieraufwand bestimmt
Da kein nicht polynomielles Problem durch die Obfuskation entsteht, bewegt sich die Resistenz im Bereich Polynomialzeit

Unterteilung in (betriff nur):
lokal: ein Basic Block; global: gesamter cfg, prozed: Informationsfluss zwischen Funktionen, prozess: zwischen Threads
global auf jeden Fall (Sinn der obfus.), auch etwas prozed., weil Datenfluss und Distanz

dementsprechend kann die Einordnung wie folgt erfolgen

Formelle Obfuskationsmetriken (Widerstandsfähigkeit)

- Aufwand zur Erstellung eines Deobfuskators



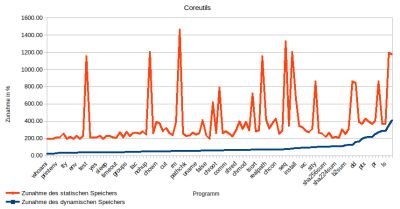
auf jeden Fall generische Deobfuskation unmöglich, weil geprüft werden müsste, ob Thread durch Obfuskation oder schon vorhanden nicht nicht obfuskierter Variante

auch wenn nur schwach bis stark, eigenständig selten umfassend, nur in Kombination

Formelle Obfuskationsmetriken (Kosten)

Transformationsdauer: Zuwachs um ca. 50 %

Speicherplatz



Performanz

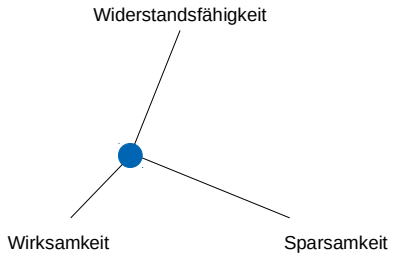
Programm	LOC	Anzahl Funktionen	Programm-größe in B	Max Zunahme in %
BBP Formel	149	5	14664	15463,89
bzip2	6419	109	99728	1758444,95
Verhältnis	43,08	21,80	6,80	113,71

Ressourcenzuwachs
konstant=kostenlos, linear=billig, polynomiell=kostspielig, exponentiell=teuer

- konnte ein durchschnittlicher Zuwachs von 50% gemessen werden, vertretbar (konstant-linear)
- statisch bereits gehabt (400%), dynamisch im Mittel: 81,27%, jedoch kein Zusammenhang, wie aus der Abbildung hervorgeht, angesichts heutiger Kapazitäten und Raten vertretbar (linear)
- Performanz gemäß Graph bei höherer Ausführungsdauer anscheinend konstant, in Abhängigkeit von der Programmgröße erscheint jedoch die Zunahme sogar expo.

alleine aufgrund der Zunahme um fast 2.000.000 % ist sind die Kosten hierdurch teuer

Formelle Obfuskationsmetriken (Zusammenfassung)



Nochmal das Spannungsdreick, in welchem versucht wurde die Obfuskationstechnik einzuordnen.

Wirksamkeit: Programmgröße ist hervorzuheben, Abzüge wegen Fokus auf Codesektion, sonst mehrere verschiedene Auswirkungen, deshalb „mittel“
Widerstand: schwach bis stark
Kosten: teuer, dementsprechend geringe Sparsamkeit

Relativierung durch die Möglichkeit durch Aussparung rechenintensiver Methoden die Einordnung zur Sparsamkeit zu verschieben. So stellt auch die Zielgruppe, insbesondere Malware, ein weniger großes Problem dar wegen Wartezeitverhältnis (Mimikatz)

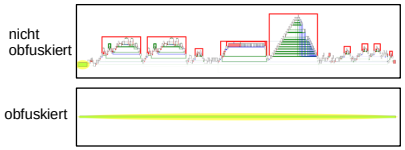
Auswirkungen auf das Reverse Engineering

Statische Analyse

- Disassemblierung/ Dekompilierung
- Kontrollflussgraph (siehe. Bild)
- Funktionsaufrufgraph

Dynamische Analyse

- Threadwechsel
- Stacktrace
- Hohe Belastung
- Zusätzliche Systemfunktionen



Nutzerstudie besser, aber im Folgenden Konsequenzen möglichst praxisnah beleuchtet

- Disas. komplexer, mehr Referenzen; Dekomp. Funktionalität verteilt und Zusammenhang der Schnipsel nicht so leicht ersichtlich
- Auslöschung jeglicher Informationen aus dem Kontrollflussgraph als Ziel erfolgreich
- aufgrund zu vieler Referenzen nicht darstellbar, zusätzliche Schicht

- umständlich und irritierend
- Stacktrace als Ursprung immer die Erstellung des Threads
- Zahlreiche Threads, in einem Beispiel mit bzip2 fast 300 aktive Threads und mehr als 9 Mio erstellt
- Zusätzlich CreateThread, Close, DuplicateObject

Auswirkungen auf die Malware Detektion

Signaturbasierte Detektion

- Lokal Sensitiver Hash

- YARA

64-bit		32-bit	
n. obfus.	**obfus.**	**n. obfus.**	**obfus.**
36	7	49	8

Verhaltensbasierte Detektion

- Cuckoo Sandbox
- Geringe Ausgangsdetektionsrate
- Starke Auswirkung bei einem Sample

Kommerzielle Scanner

- Weniger Detektionen bei mehr als 60 % der Samples
- Im Mittel 2 Detektionen weniger

Inhalt

- Nahezu keine Ähnlichkeit zwischen obfus./n.obfus, auch Distanz ähnlich beliebigen Referenzprogs, PoC bzgl. Polymorphie (Params)
- Sigs vom Yara-Signator des FKIE, Anzahl der Detektionen durch obfus. um ca 80 % reduziert
  Umstand, wie Codesektion und max. Progsize, sofern eine obfuskierte Variante im Gen-Proc, deutlich bessere Ergebnisse erwartbar
- Schlagworte; Reduktion von mehr als 5 auf unter 1, obwohl verifiziert, dass Sample innerhalb Timeout aktiv war
  Aufgrund unbeständigem Verhalten schwer die Gründe ohne umfangreiche Untersuchung
- Schlagworte=Ergebnis; Black-Box, Einzelfallbetrachtung, auch Zunahme

Heuristische Detektion basiert größtenteils auf den betrachteten Formen und wurde nur theoretisch behandelt, weshalb in dieser Ergebnispräsentation nicht näher darauf eingegangen wird

- Vergleich mit ähnlichen Publikationen
- Ansätze für Gegenmaßnahmen
- Optimierungen
  - Kombination
  - Alternativen
  - Funktionen
  - Zuverlässigkeit

Vergleich mit ähnlichen Publikationen

- Opake Konstanten mit Binary Rewriting (Moser et al.)
  - Kontrollflussgraph aufgrund indirekter Verzweigungen nicht möglich
  - Zunahme der Programmgröße vergleichbar, aber deutlich bessere Performanz
- Erzeugung von Perturbationen mit Binary Rewriting
  - Hohe Reduzierung der Detektionsraten möglich
- „Thread-basierte Obfuskation“ (Omar et al.)
  - Samples nur teilweise Vergleichbar
- Shadow Attacks: Verlagerung von Funktionalität in andere Prozesse (Ma et al.)
  - Gute Malware Evasion legitimiert gegebenenfalls hohe Kosten

Ansätze für Gegenmaßnahmen

- Gezielte Deobfuskation notwendig
  - Diagnose des Obfuskationsprinzips
  - Rückführung der Funktionalität in die ursprüngliche Funktion anhand der Systemaufrufe zum Freigeben eines Semaphors
  - Entfernung weiterer unnötiger Artefakte
- Dynamische Analyse durch Aufzeichnung der Systemaufrufe
  - Filterung der Systemaufrufe der Obfuskation

- Schlagworte; Programsize auch 400%, Performanz nur 50–100% (im Gegensatz zu 15.000% bei BBP)
- Im Schnitt um zwanzig, auch bis runter auf 8 (von ca. 60), Binary Rewriting aber im allgemeinen besser, sodass die hier erreichten 12 Detektionen nicht vernachlässigbar
- Sehr ähnlich, aber nur minimale-kmplexe Samples; Ausführungszeit annähernd vergleichbar
- Verteilung von Codeschnipsel zur Ausführung in anderen Prozessen; auch hier ähnlich hohe Zunahme der Ausführungsdauer, so wird in dieser Arbeit jedoch postuliert, dass Malware Entwickler eine gute Evasion bevorzugen, wenn nicht zeitkritische Aufgabe

generische nicht möglich; Verstehen, wie funzt; anhand der Semaphore die Reihenfolge bestimmen und zurückverlagern; weitere Änderungen (weitere unnötig gewordene Systemcalls und Datenfluss)

dynamische Analyse nach dieser stark invasiven Deobfuskation wahrscheinlich nicht mehr möglich (also nach öffentlicher Quellen, BinaryRew), deshalb ggf. nur nicht deobfuskiert durch Aufzeichnung

in beiden Fällen jedoch ggf Analyse notwendig, ob Element von Obfuskation oder vorher

Aufwand überschaubar, aber Obfuskation leicht erweiterbar und modifizierbar; Schwierigkeit die Entwicklung eines generischen Deobfuskators

Optimierungen

- Kombination (Erhöhung Wirksamkeit und Widerstandsfähigkeit)
  - Verschleierung der Zugriffe auf die Semaphore (ggf. opake Konstanten)
  - Mehrfache Nutzung von Variablen bzw. parallele, unproduktive Funktionalität
- Alternativen
  - Mehrfach durchlaufene Funktionalität per Schleife neu starten
  - Anwendung des Thread-Pool-Musters
- Funktionsumfang
  - Einschränkungen beseitigen (Ausnahmebehandlung, etc.)
  - Metamorphie
- Zuverlässigkeit
  - Behandlung der Ressourcenprobleme und weiterer Komplikationen

- die effektivsten Kombinationen; aliasing problem (mehrere Zugriffsmglkeit | indirekt) – opak lesen;mehrfache Nutzung globVar und Sem. Analyse bei Obfus auch bei Deobfus (Widerstnd) Parallel, sodass semantische Anal., auch Mimikry
- also nicht create thread sondern einfach nach Abarbeitung an den Anfang springen; Anstatt viele Threads, nur einige, die eine Funktionsptrq abarbeiten – Prüfung ergab keine signifikant Perf
- und rekursiv&variableParam; reflexive Transformation (rewrite/rückübersetzung), aber im Kontext eher Auslagerung anderer Funknalität, Substitution globVars oder change order
- während transform berechnen oder setrlimit; linux signal func

- In diesem Zusammenhang Einbeziehen von Compileroptimierungen

Inhalt

- Vergleich mit ähnlichen Publikationen
- Ansätze für Gegenmaßnahmen
- Optimierungen
  - Kombination
  - Alternativen
  - Funktionen
  - Zuverlässigkeit

Zusammenfassung und Ausblick

- Aussparung rechenintensiver Routinen
- Einteilungsgröße
- Parametrisierung
- Binary Rewriting

Ende

Vielen Dank für Ihre Aufmerksamkeit.

Grundlagen:CFG,RevEng,Obfusk.allg.,MalwareD&E
Design&Impl:nur1möglichkeit,warum,algo
Eval:Beweis.enough,formell.high.cost,reveng.praxis, mal.det partly nicht zur vernachlässigen
Disk:compare-lässt.sich.einordnen, deob.überschaubar aber erweiterbar, diverse Optimierungen, Übergang zum Ausblick

so wurde mehr breite, bildet aber die Grundlage für Spezialis. und Prioris. bei Untersuchung

Die wichtigsten Schlagwörter (und warum)
- Kosten, auch automatisierte Aussparung (hotspot)
- Wahl auch Kosten aber auch Wirksam/Wider, ggf. auch variabel
- In diesem Zsm auch Parametrisierung, aber auch bspw. reihenfolge; interessant, wie gut yara-sig mit obfusk.sample und dann param.samples
- abschließend festgestllt, dass BinRew vielerlei nutz

- uneingeschränkt, nur höchste Effektivität
-
- Einbeziehen von Compileroptimierungen

Quellen

[1] Banescu, Sebastian & Pretschner, Alexander. (2017). A Tutorial on Software Obfuscation. Advances in Computers, vol. 108, pp. 283–353, Elsevier. 10.1016/bs.adcom.2017.09.004.
[2] Collberg, Christian & Thomborson, Clark & Low, Douglas. (1997). A Taxonomy of Obfuscating Transformations.
[3] Watt, David & Findlay, William. (2004). Programming language design concepts.
[4] Intel. (2021). Intel 64 and IA-32 Architectures Software Developer's Manual.
[5] Markgraf, Daniel. (2018). Reverse Engineering. https://wirtschaftslexikon.gabler.de/definition/reverse-engineering-45260/version-268557. Letzter Zugriff 06.05.2021.
[6] Yakdan, Khaled & Dechand, Sergej & Gerhards-Padilla, Elmar & Smith, Matthew. (2016). Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. 2016 IEEE Symposium on Security and Privacy (SP), pp. 158-177. 10.1109/SP.2016.18.
[7] Ghidra. https://ghidra-sre.org/. Letzter Zugriff 18.04.2021.
[8] Eilam, E. (2005). Reversing: Secrets of Reverse Engineering.
[9] Choi, Jusop & Shin, Dongsoon & Kim, Hyoungshick & Seobs, Jason & Hong, Jin. (2019). AMVG: Adaptive Malware Variant Generation Framework Using Machine Learning. 2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC), pp. 246-24609. 10.1109/PRDC47002.2019.00055.
[10] Wenzl, Matthias & Merzdovnik, Georg & Ullrich, Johanna & Weippl, Edgar. (2019). From Hack to Elaborate Technique—A Survey on Binary Rewriting. ACM Computing Surveys 52, pp 1-37. 10.1145/3316415.
[11] Bruschi, Danilo & Martignoni, Lorenzo & Monga, Mattia. (2006). Using Code Normalization for Fighting Self-Mutating Malware.
[12] Bundesamt für Sicherheit in der Informationstechnik. https://www.bsi.bund.de/DE/Themen/Verbraucherinnen-und-Verbraucher/Informationen-und-Empfehlungen/Cyber-Sicherheitsempfehlungen/Virenschutz-Firewall/Virenschutzprogramme/virenschutzprogramme_node.html. Letzter Zugriff 06.05.2021.
[13] Christodorescu, Mihai &Jha, Somesh & Maughan, Douglas & Song, Dawn &Wang, Cliff. (2006). Malware Detection (Advances in Information Security). Springer-Verlag, Berlin, Heidelberg.
[14] Damodaran, Anusha & Di Troia, Fabio & Visaggio, Corrado Aaron & Austin, Thomas & Stamp, Mark. (2017). A comparison of static, dynamic, and hybrid analysis for malware detection. Journal of Computer Virology and Hacking Techniques 13, pp. 1-12. 10.1007/s11416-015-0261-z.
9)[15] Trend Micro. TLSH – Trend Micro Locality Sensitive Hash. https://github.com/trendmicro/tlsh. Letzter Zugriff 18.04.2021.
[16] Upchurch, Jason & Zhou, Xiaobo. (2015). Variant: a malware similarity testing framework. 2015 10th International Conference on Malicious and Unwanted Software (MALWARE), pp. 31-39. 10.1109/MALWARE.2015.7413682.
[17] VirusTotal. Yara. https://virustotal.github.io/yara/. Letzter Zugriff 18.04.2021.
[18] Bonfante, Guillaume & Kaczmarek, Matthieu & Marion, Jean-Yves. (2007). Control Flow Graphs as Malware Signatures.
[19] Bruschi, Danilo & Martignoni, Lorenzo & Monga, Mattia. (2006). Detecting Self-mutating Malware Using Control-Flow Graph Matching. Detection of Intrusions and Malware & Vulnerability Assessment. DIMVA 2006. 129-143. 10.1007/11790754_8.
[20] Campion, Marco & Dalla Preda, Mila & Giacobazzi, Roberto. (2021). Learning metamorphic malware signatures from samples. Journal of Computer Virology and Hacking Techniques, pp. 1-17. 10.1007/s11416-021-00377-7.
[21] Eskandari, Mojtaba & Hashemi, Sattar. (2012). A graph mining approach for detecting unknown malwares. Journal of Visual Languages & Computing 23, pp. 154–162. 10.1016/j.jvlc.2012.02.002.
[22] Adkins, Francis & Jones, Luke & Carlisle, Martin & Upchurch, Jason. (2013). Heuristic malware detection via basic block comparison. 2013 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE), pp. 11-18. 10.1109/MALWARE.2013.6703680.
[23] Cohen, Fred. (1990). A Short Course on Computer Viruses. Computers & Security - COMPSEC 8.
[24] Microsoft. (2021). Über die Cloud bereitgestellten Netzwerkschutz aktivieren. https://docs.microsoft.com/de-de/microsoft-365/security/defender-endpoint/enable-cloud-protection-microsoft-defender-antivirus?view=o365-worldwide. Letzter Zugriff 01.08.2021.
[25] Cuckoo. https://cuckoosandbox.org/. Letzter Zugriff 18.04.2021.
[26] Raff, Edward & Barker, Jon & Sylvester, Jared & Brandon, Robert & Catanzaro, Bryan & Nicholas, Charles. (2017). Malware Detection by Eating a Whole EXE. AAAI Workshops.
[27] Rieck, Konrad & Trinius, Philipp & Willems, Carsten & Holz, Thorsten. (2011). Automatic analysis of malware behavior using machine learning. Journal of Computer Security 19, pp. 639-668. 10.3233/JCS-2010-0410.
[28] Ahmed, Faraz & Hameed, Haider & Shafiq, M. & Farooq, Muddassar. (2009). Using spatio-temporal information in API calls with machine learning algorithms for malware detection. AISec '09: Proceedings of the 2nd ACM workshop on Security and artificial intelligence, pp. 55-62. 10.1145/1654988.1655003.
[29] Firdausi, Ivan & Lim, Charles & Erwin, Alva & Nugroho, Anto. (2010). Analysis of Machine learning Techniques Used in Behavior-Based Malware Detection. 2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies, pp. 201-203. 10.1109/ACT.2010.33.
[30] Souri, Alireza & Hosseini, Rahil. (2018). A state-of-the-art survey of malware detection approaches using data mining techniques. Human-centric Computing and Information Sciences 8, pp. 1-22. 10.1186/s13673-018-0125-x.
[31] Kaspersky. Machine Learning Methods for Malware Detection. (2020). https://media.kaspersky.com/en/enterprise-security/Kaspersky-Lab-Whitepaper-Machine-Learning.pdf. Letzter Zugriff 18.04.2021.
[32] Sophos. Machine Learning: How to Build a Better Thread Detection Model. (2017). https://www.sophos.com/en-us/medialibrary/PDFs/technical-papers/machine-learning-how-to-build-a-better-threat-detection-model.pdf. Letzter Zugriff 18.04.2021.
[33] Zhang, Qinghua & Reeves, D.s. (2007). MetaAware: Identifying Metamorphic Malware. Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), pp. 411-420. 10.1109/ACSAC.2007.9.
[34] Wong, Wing & Stamp, Mark. (2006). Hunting for metamorphic engines. Journal in Computer Virology 2, pp. 211-229. 10.1007/s11416-006-0028-7.
[35] Marpaung, Jonathan & Sain, Mangal & Lee, Hoon-Jae. (2012). Survey on malware evasion techniques: State of the art and challenges. 2012 14th International Conference on Advanced Communication Technology (ICACT), pp. 744-749.

- uneingeschränkt, nur höchste Effektivität
-
- Einbeziehen von Compileroptimierungen

Quellen

[36] Afianian, Amir & Niksefat, Salman & Sadeghiyan, Babak & Baptiste, David. (2019). Malware Dynamic Analysis Evasion Techniques: A Survey. ACM Computing Surveys 52, pp. 1-28. 10.1145/3365001.
[37] UPX. https://upx.github.io/. Letzter Zugriff 18.04.2021.
[38] Reece, Alex. (2013). Introduction to return oriented programming (ROP). https://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html. Letzter Zugriff 20.05.2021.
[39] Demetrio, Luca & Biggio, Battista & Lagorio, Giovanni & Roli, Fabio & Armando, Alessandro. (2019). Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries.
[40] Rosenberg, Ishai & Gudes, Ehud. (2016). Bypassing system calls-based intrusion detection systems. Concurrency and Computation: Practice and Experience. 29. 10.1002/cpe.4023.
[41] Li, Xufang & Loh, Peter & Tan, Freddy. (2011). Mechanisms of Polymorphic and Metamorphic Viruses. 2011 European Intelligence and Security Informatics Conference, pp. 149-154. 10.1109/EISIC.2011.77.
[42] Borello, Jean-Marie & Mé, Ludovic. (2008). Code Obfuscation Techniques for Metamorphic Viruses. Journal in Computer Virology 4, pp. 211-220. 10.1007/s11416-008-0084-2.
93[43] Murad, Khurram & Shirazi, Syed & Zikria, Yousaf & Ikram, Nassar. (2010). Evading Virus Detection Using Code Obfuscation. Future Generation Information Technology, pp. 394-401. 10.1007/978-3-642-17569-5_39.
[44] Payer, Mathias. (2014). Embracing the new threat: towards automatically, self-diversifying malware.
[45] Junod, Pascal & Rinaldini, Julien & Wehrli, Johan & Michelin, Julie. (2015). Obfuscator-LLVM -- Software Protection for the Masses. 2015 IEEE/ACM 1st International Workshop on Software Protection, pp. 3-9. 10.1109/SPRO.2015.10.
[46] Larsen, Per & Homescu, Andrei & Brunthaler, Stefan & Franz, Michael. (2014). SoK: Automated Software Diversity. 2014 IEEE Symposium on Security and Privacy, pp. 276-291. 10.1109/SP.2014.25.
[47] Veerappan, Chandra Sekar & Keong, Peter & Tang, Zhaohui & Tan, Forest. (2018). Taxonomy on malware evasion countermeasures techniques. 2018 IEEE 4th World Forum on Internet of Things (WF-IoT), pp. 558-563. 10.1109/WF-IoT.2018.8355202.
[48] Yuval tisf Nativ. TheZoo – A Live Malware Repository. https://github.com/ytisf/theZoo. Letzter Zugriff 18.04.2021.
[49] Ma, Weiqin & Duan, Pu & Liu, Sanmin & Gu, Guofei & Liu, Jyh-charn. (2011). Shadow attacks: Automatically evading system-call-behavior based malware detection. Journal in Computer Virology 8, pp. 1-13. 10.1007/s11416-011-0157-5.
[50] Omar, Rasha & El-Mahdy, Ahmed & Rohou, Erven. (2013). Thread-Based Obfuscation through Control-Flow Mangling.
[51] LLVM- The LLVM Compiler Infrastructure. https://llvm.org/. Letzter Zugriff 18.04.2021.
[52] Statista. (2021). Marktanteile der führenden Betriebssysteme weltweit im Oktober 2020. https://de.statista.com/statistik/daten/studie/626810/umfrage/marktanteile-der-fuehrenden-betriebssystemversionen-weltweit/. Letzter Zugriff 20.05.2021.
[53] Andrzej Warzynski. How to create pass independently on Windows. (2020). https://llvm.discourse.group/t/how-to-create-pass-independently-on-windows/474. Letzter Zugriff 18.04.2021.
[54] LLVM. Clang: a C language family frontend for LLVM. https://clang.llvm.org/. Letzter Zugriff 18.04.2021.
[55] Statista. (2021). Die beliebtesten Programmiersprachen weltweit laut PYPL-Index im März 2021. https://de.statista.com/statistik/daten/studie/678732/umfrage/beliebteste-programmiersprachen-weltweit-laut-pypl-index/. Letzter Zugriff 20.05.2021.
[56] Sutter, Herb & Alexandrescu, Andrei. (2004). C++ Coding Standards.
[57] LLVM. Exception Handling in LLVM. https://llvm.org/docs/ExceptionHandling.html. Letzter Zugriff 18.04.2021.
94[58] Linux man page. pthread_mutex_lock(3). (2003). https://linux.die.net/man/3/pthread_mutex_lock. Letzter Zugriff 18.04.2021.
[59] Linux manual page. pthread_spin_lock(3). (2021). https://man7.org/linux/man-pages/man3/pthread_spin_lock.3.html. Letzter Zugriff 18.04.2021.
[60] Linux manual page. (2021). pthread_cancel(3). https://man7.org/linux/man-pages/man3/pthread_cancel.3.html. Letzter Zugriff 05.07.2021.
[61] Microsoft. (2018). TerminateThread function. https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-terminatethread. Letzter Zugriff 05.07.2021.
[62] Microsoft. (2018). Slim ReaderWriter (SRW) Locks. https://docs.microsoft.com/en-us/windows/win32/sync/slim-reader-writer--srw-locks. Letzter Zugriff 12.07.2021.
[63] Trail of Bits. McSema. https://github.com/lifting-bits/mcsema. Letzter Zugriff 18.04.2021.
[64] Hex Rays. IDA Pro. https://www.hex-rays.com/products/ida/. Letzter Zugriff 18.04.2021.
[65] Petr Zemek. Compiling back LLVM IR. https://github.com/avast/retdec/issues/45. Letzter Zugriff 18.04.2021.
[66] mk-mode. C++ source code to compute pi with BBP formula. https://gist.github.com/komasaru/9e418eb66ab649ef589. Letzter Zugriff 18.04.2021.
[67] Julian Seward. bzip2. https://github.com/opencor/bzip2. Letzter Zugriff 18.04.2021.
[68] GNU. Coreutils – GNU core utilities. (2020). https://www.gnu.org/software/coreutils/. Letzter Zugriff 28.06.2021.
[69] Moser, Andreas & Kruegel, Christopher & Kirda, Engin. (2008). Limits of Static Analysis for Malware Detection. Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), pp. 421-430. 10.1109/ACSAC.2007.21.
[70] Ding, Fei. (2017). IoT Malware. https://github.com/fding/iot-malware. Letzter Zugriff 03.05.2021.
[71] Rapid7. Mettle. https://github.com/rapid7/mettle. Letzter Zugriff 04.06.2021.
[72] Karnik, Abhishek & Goswami, Suchandra. (2007). Detecting Obfuscated Viruses Using Cosine Similarity Analysis. First Asia International Conference on Modelling & Simulation (AMS'07), pp. 165-170. 10.1109/AMS.2007.31.
[73] Shabtai, Asaf & Moskovitch, Robert & Feher, Clint & Dolev, SHlomi & Elovici, Yuval. (2012). Detecting unknown malicious code by applying classification techniques on OpCode patterns. Security Informatics 1, pp. 1-22. 10.1186/2190-8532-1-1.
[74] Kolter, Jeremy & Maloof, Marcus. (2006). Learning to Detect and Classify Malicious Executables in the Wild. Journal of Machine Learning Research 7, pp. 2721-2744.
99[75] Graham Cluley. VX Heaven, old-school virus-writing website, returns from the dead after police raid. https://grahamcluley.com/vx-heaven/. Letzter Zugriff 28.05.2021.
[76] VX Heaven. VX Heaven Virus Collection 2010-05-18. https://academictorrents.com/details/34ebe49a48aa532deb9c0dd08a08a017aa04d810/tech. Letzter Zugriff 18.04.2021.
[77] Rapid7. Windows Native C meterpreter. https://github.com/rapid7/metasploit-payloads/tree/master/c/meterpreter. Letzter Zugriff 28.06.2021.

- uneingeschränkt, nur höchste Effektivität
-
- Einbeziehen von Compileroptimierungen

Quellen

[78] Benjamin Delpy. mimikatz. https://github.com/gentilkiwi/mimikatz. Letzter Zugriff 28.06.2021.
[79] Nick Allen, Tyler Failther, Laine Rumreich, Collin Thomas, Nathan Simpson, Patrick Walter. Polymorphic Virus & Countermeasures. (2019). https://github.com/LaineRumreich/PolymorphicVirus. Letzter Zugriff 28.06.2021.
[80] Stephen Fewer. ReflectiveDLLInjection. (2013). https://github.com/stephenfewer/ReflectiveDLLInjection. Letzter Zugriff 28.06.2021.
[81] Linux manual page. (2021). diff(1). https://man7.org/linux/man-pages/man1/diff.1.html. Letzter Zugriff 28.06.2021.
[82] Pádraig Brady. (2017). How the GNU coreutils are tested. http://www.pixelbeat.org/docs/coreutils-testing.html. Letzter Zugriff 14.07.2021.
[83] Linux manual page. (2021). signal(2). https://man7.org/linux/man-pages/man2/signal.2.html. Letzter Zugriff 05.07.2021.
[84] Omar, Rasha & El-Mahdy, Ahmed & Rohou, Erven. (2014). Arbitrary Control-Flow Embedding into Multiple Threads for Obfuscation: A Preliminary Complexity and Performance Analysis. SCC '14: Proceedings of the 2nd international workshop on Security in cloud computing. pp. 51-58. 10.1145/2600075.2600080.
[85] Gusfield, Dan. (1997). Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge, UK: Cambridge University Press.
[86] Coco, Moreno & Keller, Frank. (2012). Scan Patterns Predict Sentence Production in the Cross-Modal Processing of Visual Scenes. Cognitive science, vol. 36, pp. 1204-1023. 10.1111/j.1551-6709.2012.01246.x.
[87] Haq, Irfan Ul & Caballero, Juan. (2021). A Survey of Binary Code Similarity. ACM Computing Surveys, vol. 54, pp. 1-38. 10.1145/3446371.
[88] Linux Kernel Organization. Perf: Linux profiling with performance counters. (2020). https://perf.wiki.kernel.org/index.php/Main_Page. Letzter Zugriff 20.07.2021.
[89] Denis Bakhvalov. (2019). How to get consistent results when benchmarking on Linux?". https://easyperf.net/blog/2019/08/02/Perf-measurement-environment-on-Linux#4- set-cpu-affinity. Letzter Zugriff 22.07.2021.
[90] Linux manual page. (2019). time(1). https://man7.org/linux/man-pages/man1/time.1.html. Letzter Zugriff 23.07.2021.
96[91] GNU. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/. Letzter Zugriff 18.04.2021.
[92] x64dbg. https://x64dbg.com/. Letzter Zugriff 18.04.2021.
[93] GNU. GCC, the GNU Compiler Collection. https://gcc.gnu.org/. Letzter Zugriff 18.04.2021.
[94] VirusTotal. Conributors. https://support.virustotal.com/hc/en-us/articles/115002146809-Contributors. Letzter Zugriff 10.08.2021.
[95] Bilstein, Felix & Plohmann, Daniel. (2019). YARA-Signator: Automated Generation of Code-based YARA Rules. The Journal on Cybercrime & Digital Investigations, vol. 5, no. 1. Botconf 2019 Proceedings. 10.18464/cybin.v5i1.24.
[96] Sulaiman, A. & Ramamoorthy, K. & Mukkamala, Srinivas & Sung, Andrew. (2005). Malware examiner using disassembled code (MEDIC). Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop, pp. 428-429. 10.1109/IAW.2005.1495985.
[97] Oracle. VirtualBox. https://www.virtualbox.org/. Letzter Zugriff 18.04.2021.
[98] Thomas Hungenberg & Matthias Eckert. (2020). InetSim: Internet Services Simulation Suite. https://www.inetsim.org/. Letzter Zugriff 29.07.2021.
[99] Qiao, Yong & He, Jie & Yang, Yuexiang & Ji, Lin. (2013). Analyzing Malware by Abstracting the Frequent Itemsets in API Call Sequences. 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, pp. 265-270. 10.1109/TrustCom.2013.36.
[100] Dai, Jianyong & Joohan, Lee. (2009). Efficient Virus Detection Using Dynamic Instruction Sequences. Journal of Computers 4, pp. 405-414. 10.4304/jcp.4.5.405-414.
[101] Wendelin, Claes Adam. (2017). Malware Detection in Secured Systems. Seminar Future Internet SS2017. 10.2313/NET-2017-09-1_11.
[102] Sorokin, Ivan. (2011). Comparing files using structural entropy. Journal in Computer Virology 7, pp. 259-265. 10.1007/s11416-011-0153-9.
[103] Alexander H. Liu. (2018). MalConv-Pytorch. https://github.com/Alexander-H-Liu/MalConv-Pytorch. Letzter Zugriff 18.07.2021.
[104] Kondrad Rieck. Malheur Dataset. https://www.sec.cs.tu-bs.de/data/malheur/. Letzter Zugriff 29.07.2021.
[105] VirusTotal. https://www.virustotal.com/. Letzter Zugriff 18.04.2021.
[106] Sharif, Mahmood & Lucas, Keane & Bauer, Lujo & Reiter, Michael & Shintre, Saurabh. (2019). Optimization-Guided Binary Diversification to Mislead Neural Networks for Malware Detection.
[107] Castro, Raphael & Schmitt, Corinna & Rodosek, Gabi. (2019). ARMED: How Automatic Malware Modifications Can Evade Static Detection?. 2019 5th International Conference on Information Management (ICIM), pp. 20-27. 10.1109/INFOMAN.2019.8714698.
97[108] Demetrio, Luca & Biggio, Battista & Lagorio, Giovanni & Roli, Fabio & Armando, Alessandro. (2020). Functionality-preserving Black-box Optimization of Adversarial Windows Malware. IEEE Transactions on Information Forensics and Security. 10.1109/TIFS.2021.3082330.
[109] Rossow, Christian & Dietrich, Christian & Grier, Chris & Kreibich, Christian & Paxson, Vern & Pohlmann, Norbert & Bos, Herbert & van Steen, Maarten. (2012). Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. 2012 IEEE Symposium on Security and Privacy. pp. 65-79. 10.1109/SP.2012.14.
[110] Hu, Weiwei & Tan, Ying. (2017). Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN.
[111] Pavithran, Jithin & Patnaik, Milan & Rebeiro, Chester. (2019). D-TIME: Distributed Threadless Independent Malware Execution for Runtime Obfuscation. WOOT @ USENIX Security Symposium.
[112] Rosenberg, Ishai & Shabtai, Asaf & Rokach, Lior & Elovici, Yuval. (2018). Generic Black-Box End-to-End Attack against RNNs and Other API Calls Based Malware Classifiers. Research in Attacks, Intrusions, and Defenses. 10.1007/978-3-030-00470-5_23.
[113] Zhou, Ruuyu. (2018). Guided Automatic Binary Parallelisation. 10.17863/CAM.21890.
[114] Wikipedia. Pointer analysis. https://en.wikipedia.org/wiki/Pointer_analysis. Letzter Zugriff 18.04.2021.
[115] Wikipedia. Längster Pfad. https://de.wikipedia.org/wiki/L%C3%A4ngster_Pfad. Letzter Zugriff 18.04.2021.
[116] Linux manual page. GetrlimIt(2). (2021). https://man7.org/linux/man-pages/man2/setrlimit.2.html. Letzter Zugriff 19.05.2021.
[117] LLVM. GlobalOpt.cpp. https://github.com/llvm/llvm-project/blob/llvmorg-11.1.0/llvm/lib/Tranisforms/IPO/GlobalOpt.cpp. Letzter Zugriff 18.04.2021.

- uneingeschränkt, nur höchste Effektivität
-
- Einbeziehen von Compileroptimierungen