

Automatisierte Signaturgenerierung für Malware-Stämme

Christian Blichmann¹ · Thomas Dullien² · Michael Meier³

¹zynamics GmbH
christian.blichmann@zynamics.com

²zynamics GmbH
thomas.dullien@zynamics.com

³TU Dortmund
michael.meier@udo.edu

Zusammenfassung

Eine ständig wachsende Zahl von Schadprogrammen für Rechensysteme, auch als Malware bezeichnet, stellt Antivirenprogramme vor die Herausforderung, diese möglichst schnell und mit niedrigen Fehleraten zu erkennen. Eine große Anzahl zu überprüfender Signaturen führt hierbei zu einem Trade-Off zwischen hoher Erkennungsrate auf der einen und möglichst guten Antwortzeiten des Rechensystems auf der anderen Seite. Aktuelle Signaturen sind für die Sicherheit moderner Rechensysteme jedoch unverzichtbar. Im Rahmen dieser Arbeit wird eine Methode vorgestellt, mit der herkömmliche Scanner von – für Echtzeiteinsatz häufig nicht geeigneten – automatischen Klassifikationsmethoden für Malware profitieren können. Hierbei wird für eine gegebene Familie von Schadprogrammen eine Byte-basierte Signatur für den Scanner ClamAV generiert, mit der zuverlässig die gesamte Familie erkannt werden kann. Die so erzeugte Signatur wird anschließend in Bezug auf falsche Positive und falsche Negative getestet.

1 Einführung

Malware stellt für die Sicherheit der Infrastruktur einer Informationsgesellschaft eine ernstzunehmende Bedrohung dar. Unter *Malware* werden hierbei allgemein unerwünschte Programme auf einem Rechensystem zusammengefasst. Hierzu zählen Würmer, Viren und Trojanische Pferde, sowie als Spyware bezeichnete Programme und Programmfragmente [Ayco06].

In der Praxis ist eine ständig wachsende Zahl von Schadprogrammen zu beobachten, die Antivirenprogramme vor die Herausforderung stellt, diese möglichst schnell und mit niedrigen Fehleraten zu erkennen. Häufig wird das syntaktische Erscheinungsbild eines bereits existierenden Schadprogramms variiert, um eine Entdeckung zu erschweren. Damit einhergehend steigt die Zahl der von Antivirenprogrammen zu überprüfenden Signaturen, was zu einem Trade-Off zwischen hoher Erkennungsrate auf der einen und möglichst guten Antwortzeiten des Rechensystems auf der anderen Seite führt.

Das Problem spiegelt sich auch in der zunehmenden Zahl von Duplikaten in den Malware-Datenbanken der Hersteller von Antivirenprogrammen wider. Laut einem Bericht des Anbieters McAfee [GrTe07] wurden 2007 von ca. 2,8 Mio. gesammelten Malware-Exemplaren 1,7 Mio. als

Duplikate erkannt. Insgesamt wird der Anteil der Duplikate in der Malware-Datenbank auf etwa 36 % geschätzt. Seitens der Industrie bedeutet eine zahlenmäßige Zunahme von Malware einen immensen personellen Aufwand bei der Analyse der jeweiligen Schadprogramme und der – meist manuellen – Erstellung von Signaturen. Ebenfalls nicht zu vernachlässigen ist der finanzielle Aufwand, der für die Verteilung der Signaturen und der dafür benötigten Netzwerk-Ressourcen anfällt. Da aktuelle Signaturen für die Sicherheit moderner Rechensysteme jedoch unverzichtbar sind, verspricht eine weitgehende Automatisierung des Signaturerstellungsprozesses eine drastische Reduzierung von Aufwand und Kosten. Weiterhin führt eine verminderte Anzahl von durch Antivirenprogrammen zu überprüfender Signaturen direkt zu besseren Antwortzeiten auf den eingesetzten Rechensystemen.

Manuelle Analysen von Malware (zum Beispiel mit IDA Pro [SA08]) zeigen, dass Varianten einer Malware trotz unterschiedlichen syntaktischen Erscheinungsbildes semantisch meist nahezu identisch sind. Die Erzeugung von Varianten erfolgt zumeist mit Techniken wie *Polymorphie* oder *Metamorphie* [SzFe01], die den Objektcode des jeweiligen Schadprogramms verändern. In Anlehnung an die Biologie werden Varianten einer Malware, die durch Modifikationen des Objektcode erzeugt wurden, als *Stamm* oder *Familie* bezeichnet.

Mit VxClass [GmbHa] existiert ein Werkzeug, das – ausgehend von der Analyse struktureller Ähnlichkeiten von Programmen - Malware automatisiert Familien zuordnen kann.

Die Zahl der von einem Antiviren-Programm zu verwendenden Signaturen kann signifikant verringert werden, wenn es gelingt zu einer Familie von gegebener Malware eine einzelne *Familiensignatur* zu generieren, mit der alle Samples der Familie erkannt werden können. Die Erstellung einer solchen Familiensignatur entspricht der Lösung des *k-Longest-Common-Subsequence-Problems* (k-LCS) über der Menge der syntaktischen Repräsentationen aller gegebenen *k* Samples einer Familie. Algorithmen zur Lösung dieses im allgemeinen Fall NP-harten Problems sind bekannt, kommen jedoch auf Grund ihrer Laufzeiteigenschaft für den praktischen Gebrauch nicht in Betracht (vgl. [Maie78]).

In diesem Beitrag wird ein neuartiges Verfahren vorgestellt, dass die automatische Erstellung von Familiensignaturen erlaubt. Es nutzt die oben erwähnten strukturellen Ähnlichkeiten von Samples einer Familie aus und fokussiert zur Signaturgenerierung nur auf die Instruktionsfolgen in der syntaktischen Repräsentation der Malware-Samples, die den identifizierten strukturellen Ähnlichkeiten entsprechen und im Folgenden als *Basic-Blocks* bezeichnet werden. Basic-Blocks können in den einzelnen Malware-Samples in unterschiedlicher Reihenfolge auftreten. Es muss also die Menge der Blöcke ermittelt werden, die in allen Malware-Samples in der gleichen Reihenfolge auftreten. Bei *n* Basic-Blocks kann deren relative Position in den Samples als *n*-stellige Permutation aufgefasst werden. Entsprechend bleibt das k-LCS-Problem auf Eingaben zu lösen, die diese Permutationseigenschaft erfüllen. Es wurde ein Algorithmus zur Lösung dieses Problems entwickelt, der durch Ausnutzung dieser Eigenschaft signifikant bessere Laufzeiteigenschaften als Algorithmen zur Lösung des allgemeinen k-LCS-Problems aufweist und damit für typische Eingabegrößen praktikabel ist.

Auf der Grundlage des entwickelten Algorithmus wurde ein Werkzeug zur automatischen Erstellung von Familien-Signaturen für den Antiviren-Scanner ClamAV [Clam08] implementiert und hinsichtlich seiner Praxistauglichkeit experimentell evaluiert.

2 Automatische Klassifizierung von Malware anhand struktureller Ähnlichkeiten

Das Erstellen von Signaturen für Stämme von Malware erfordert zuerst die Einordnung der ausführbaren Dateien einer gegebenen Menge von Schadprogrammen in die jeweiligen Familien. Allgemein wird die Einordnung von ausführbaren Dateien in eine Familie als *Klassifikation*

bezeichnet. Für eine Klassifizierung einer Menge von Schadprogrammen anhand struktureller Merkmale ist also eine Methode für den strukturellen Vergleich zweier ausführbaren Dateien erforderlich, wie sie etwa in dem Softwarewerkzeug BinDiff [GmbH] implementiert wird.

Eine ausführbare Datei A besteht aus einer Menge $F_{(A)} = \{f_{(A),1}, \dots, f_{(A),n}\}$ von Funktionen besteht. Der *Aufrufgraph* (*call graph*, CG) von A ist ein gerichteter Multigraph $G_{(A)} = (F_{(A)}, E_{(A)})$ mit der Menge der Funktionen $F_{(A)}$ als Knotenmenge und der Kantenmenge $E_{(A)} \subset F_{(A)} \times F_{(A)}$. Eine Kante zwischen zwei Funktionen $f_{(A),i}$ und $f_{(A),j}$ impliziert, dass $f_{(A),i}$ einen Unterprogrammaufruf von $f_{(A),j}$ enthält. Die Funktionen $f_{(A),i} \in F_A$ werden wiederum als gerichtete Graphen aufgefasst und *Kontrollflussgraphen* (*control flow graph*, CFG) genannt. Der CFG einer Funktion $f_{(A),i}$ ist gegeben durch $G_{(A),i} = (B_{(A),i}, E_{(A),i})$ mit Knotenmenge $B_{(A),i} = \{b_{(A),i,1}, \dots, b_{(A),i,m}\}$ und Kantenmenge $E_{(A),i} \subset B_{(A),i} \times B_{(A),i}$. Die Elemente $b_{(A),i,j}$ der Knotenmenge werden als *Basic Blocks* bezeichnet. Eine Kante zwischen zwei Basic-Blocks $b_{(A),i,j}$ und $b_{(A),i,k}$ impliziert eine lokale Verzweigung im Programm. Schließlich besteht ein Basic-Block $b_{(A),i,j} = \{i_{(A),i,j,1}, \dots, i_{(A),i,j,l}\}$ aus einer Folge von Instruktionen ohne Verzweigung. Da Adressen innerhalb einer ausführbaren Datei eindeutig sind, werden Funktionen, Basic-Blocks und Instruktionen jeweils mit ihren Startadressen identifiziert. Eine ausführbare Datei kann also als hierarchische gerichteter Graph aufgefasst werden, dessen Elemente CGs, CFGs und Instruktionsfolgen¹ sind.

Der strukturelle Vergleich zweier gegebener ausführbarer Dateien A und B mit $|F_{(B)}| \leq |F_{(A)}|$ könnte z.B. durch Suche nach einer Einbettung des Aufrufgraphen $G_{(B)}$ in $G_{(A)}$ erfolgen. Da das Finden solcher Einbettungen algorithmisch aufwändig ist, wird in BinDiff nach Graph-Isomorphismen gesucht.

Definition 2.1 (Graph-Isomorphismus). Zwei Graphen $G = (V, E)$ und $H = (V', E')$ heißen *isomorph* zueinander, falls eine bijektive Abbildung $\varphi : V \rightarrow V'$ existiert, so dass für alle $(e, f) \in E$ mit $e \neq f$ gilt: $(\varphi(e), \varphi(f)) \in E' \Leftrightarrow (e, f) \in E$.

Zunächst gilt: Sind zwei Dateien strukturell identisch, existiert ein Isomorphismus $\varphi : F_{(A)} \rightarrow F_{(B)}$, der die Knoten und Kanten der beiden Aufrufgraphen einander zuordnet, sowie weitere Isomorphismen, die dieselbe Funktion für die CFGs erfüllen. In der Praxis kommt dieser Fall allerdings höchst selten vor, da selbst sehr ähnliche ausführbare Dateien meist eine unterschiedliche Anzahl von Funktionen aufweisen. Die direkte Verwendung von Graph-Isomorphismen eignet sich also nur bedingt für den Vergleich von ausführbaren Dateien. BinDiff verwendet daher eine iterative Konstruktion der Graph-Isomorphismen über heuristisch definierten Eigenschaften. Für eine ausführliche Diskussion der verwendeten Algorithmen sei auf [DuRo05] und [Flak04] verwiesen.

2.1 VxClass

Aufbauend auf BinDiff ermöglicht das Softwarewerkzeug VxClass die automatisierte Klassifikation von Schadprogrammen: Die ausführbare Datei eines neu zu klassifizierenden Schadprogramms wird in einem Emulator zur Ausführung gebracht. Durch die Ausführung wird dafür gesorgt, dass eine möglicherweise vorhandene Dekodieroutine die Nutzlast dekodiert. Nach einer festgelegten Anzahl von Instruktionen wird die Ausführung gestoppt und ein Speicherabbild des Emulators zur weiteren statischen Analyse gespeichert. Nach dem Disassemblieren des erzeugten Speicherabbilds wird eine verkürzte Version von BinDiff ausgeführt. Die Malware wird so mit den bereits untersuchten Schadprogrammen verglichen. Das Ergebnis dieses Vergleichs ist ein Vektor, dessen Elemente den Grad der Ähnlichkeit der untersuchten Malware zu allen anderen verglichenen Schadprogrammen angeben. Schließlich wird die Menge der Ähnlichkeitsvektoren

¹ Eine Instruktionsfolge kann als primitiver, gerichteter, azyklischer Graph aufgefasst werden. Die Knoten der einzelnen Instruktionen werden ihrer Reihenfolge entsprechend mit Kanten verbunden.

```

1 Trojan.Crypted-3:1:EP+0:68??????00e8?????00*68000000008b74242c89e581ecc00
2 0000089e70375008a06{-8}0fb6c0

```

Listing 1: Erweiterte Signatur für eine Variante des Trojanischen Pferds Crypted

in eine Ähnlichkeitsmatrix überführt und eine Clusteranalyse durchgeführt. Das Ergebnis bestimmt die Familienzugehörigkeit der Malware. Die so durchgeführte Klassifikation mit VxClass zerlegt eine gegebene Menge von Schadprogrammen disjunkt in Familien.

3 Automatische Generierung von Familiensignaturen

3.1 ClamAV-Signaturen

ClamAV [Clam08] stellt eine Besonderheit unter den Antivirenprogrammen dar: Es ist das einzige frei im Quelltext erhältliche, nicht-kommerzielle Antivirenprogramm auf dem Markt. Signaturen für ClamAV sind in einer Signaturdatenbank für verschiedene Signaturformate organisiert. Im Folgenden wird auf Grund seiner Ausdrucksmächtigkeit nur das sogenannte erweiterte Signaturformat verwendet, weitere unterstützte Formate werden in [Kojm09] erläutert. Listing 1 zeigt ein Beispiel für das erweiterte Signaturformat. Darin enthalten sind ein Bezeichner für die Malware, die Angabe auf welche Dateitypen die Signatur anzuwenden ist, ein Offset innerhalb der Datei (auch relativ zu einzelnen Abschnitten) und eine hexadezimale Signatur. Optional kann außerdem die minimal erforderliche Scanner-Version angegeben werden. *Hexadezimale Signaturen* sind in ClamAV reguläre Ausdrücke über Byte-Sequenzen in denen jedes Byte zur Basis 16 und mit führenden Nullen dargestellt wird, wobei die üblichen alphanumerische Zeichen $\{0, \dots, 9, a, \dots, f\}$ als Ziffern verwendet werden. Als Metazeichen werden unter anderem $*$, $?$, $\{, \}$, $-$ und $-$ verwendet; Wildcards werden wie folgt dargestellt: Ein einzelnes beliebiges Byte wird von dem Ausdruck $\{?\}$ akzeptiert, eine beliebige Anzahl beliebiger Bytes durch $\{*\}$. $\{n\}$ akzeptiert genau n , $\{-n\}$ höchstens n und $\{n-\}$ mindestens n beliebige Bytes. Die disjunkte Vereinigung zweier Ausdrücke R_1 und R_2 wird in ClamAV-Syntax $\{(R_1 | R_2)\}$ geschrieben. $\{(aa | bb)\}$ akzeptiert also entweder das Byte aa oder das Byte bb .

3.2 Ziele

Ein automatisiertes Verfahren zur Generierung von Signaturen sollte folgende Ziele verfolgen: 1. Schadprogramme werden vorab automatisiert in Familien klassifiziert. 2. Das Finden von ähnlichen Fragmenten von Objektcode geschieht ohne jeden Benutzereingriff. 3. Die Erzeugung der Signaturen erfolgt auf Basis der Instruktionsbytes der identifizierten ähnlichen Stellen. 4. Es wird eine gültige und möglichst kurze Signatur erzeugt. Wildcards werden automatisch hinzugefügt. Außerdem ist es wünschenswert, dass keine oder nur wenige falsche Positive und falsche Negative bei der Verwendung der Signatur auftreten.

3.3 Verfahren

Nachfolgend wird ein Verfahren beschrieben, das die in 3.2 genannten Ziele erfüllt. Hierfür sind jedoch noch einige Definitionen erforderlich.

Definition 3.1 (String, Sequenz, Teilsequenz). Eine *Sequenz* x , auch *String* genannt, über einem endlichen Alphabet Σ ist eine geordnete Menge von Symbolen aus Σ , wobei Wiederholungen erlaubt sind, das heißt $x := \langle x_i | \forall 1 \leq i \leq n : x_i \in \Sigma \rangle$. Statt $\langle x_1, \dots, x_n \rangle$ wird kürzer auch $x_1 \dots x_n$ geschrieben und x mit einem Wort aus Σ^* identifiziert. Die *Länge* einer Sequenz wird

mit $|x|$ bezeichnet. Eine Sequenz $y = y_1 \dots y_m$ wird genau dann *Teil-* oder *Subsequenz* einer Sequenz x genannt, wenn eine Einbettung von Indizes $I = (i_1, \dots, i_m)$ mit $1 \leq i_1 < \dots < i_m \leq |x|$ aus y in x existiert, so dass für alle Indizes k mit $1 \leq k \leq m$ gilt: $x_{i_k} = y_k$. Man erhält die Teilsequenz y aus x durch Löschen von $n - m$ Symbolen aus x .

Die Menge aller Teilsequenzen von x wird mit $\text{seq}(x) := \{y \mid y \text{ ist Teilsequenz von } x\}$ bezeichnet. Für eine Teilsequenz kann es mehrere mögliche Einbettungen geben. Beispielsweise ist `aar` eine Teilsequenz von `aardvark`, mit den vier möglichen Einbettungen $(1, 2, 3)$, $(1, 2, 7)$, $(1, 6, 7)$ und $(2, 6, 7)$.

Definition 3.2 (*k*-Longest Common Subsequence Problem). Eine Sequenz x heißt *gemeinsame Teilsequenz* einer gegebenen Menge von Sequenzen $\mathcal{S} = \{S_{(1)}, \dots, S_{(k)}\}$, wenn sie eine Teilsequenz von allen Sequenzen aus \mathcal{S} ist, also $\forall S_{(i)} \in \mathcal{S} : x \in \text{seq}(S_{(i)})$ gilt. Eine *längste gemeinsame Teilsequenz* $\text{LCS}(\mathcal{S})$ von \mathcal{S} ist eine Teilsequenz maximaler Länge: $\text{LCS}(\mathcal{S}) := \max_{S_{(i)} \in \mathcal{S}} \{|x| \mid x \in \text{seq}(S_{(i)})\}$. Das Finden einer längsten gemeinsamen Teilsequenz einer Menge von k Sequenzen wird als *k-Longest Common Subsequence Problem* (kurz *k-LCS*) bezeichnet. Für den Fall $k = 2$ wird auch von dem *Longest Common Subsequence Problem* (LCS) gesprochen.

Eingabe: Eine Familie $M = \{M_1, \dots, M_n\}$ von Schadprogrammen.

Ausgabe: Eine ClamAV-Signatur S für die Schadprogramme aus M .

Vorab wird zur Klassifizierung einer Menge von Schadprogrammen das Softwarewerkzeug VxClass benutzt. Das Ergebnis der Vorverarbeitung ist eine Zerlegung der Ausgangsmenge in Familien, aus der eine Familie M für die Generierung einer Signatur ausgewählt wird.

1. Die einzelnen Schadprogramme aus M werden in einer beliebigen, aber festen Reihenfolge angeordnet und BinDiff auf alle Paare (M_i, M_{i+1}) mit $0 < i < n$ angewendet. Dies liefert jeweils $d = n - 1$ Mengen von Adresspaaren der zwischen M_i und M_{i+1} gefundenen strukturell identischen Funktionen und Basic-Blocks.
2. Es wird eine Menge von *Kandidatenfunktionen* berechnet. Dies sind Funktionen, die strukturell in allen M_i vorkommen.
3. Aus den Basic-Blocks der Kandidatenfunktionen werden nun analog zu 2. *Kandidaten-Basic-Blocks* berechnet, die in allen Elementen der Malware-Familie vorkommen. Dabei wird eine $c \times d$ -Matrix $B_{cand} = (b_{i,j})$, die die Adressen der einzelnen Kandidaten-Basic-Blocks festhält, angelegt, wobei c die Anzahl der gefundenen Kandidaten-Basic-Blocks, ergibt bezeichnet.
4. Für jedes Schadprogramm M_i wird nun das „Wort“ über dem Alphabet der Kandidaten-Basic-Blocks generiert. Anders ausgedrückt: Für jedes M_i wird eine Permutation π_i bestimmt, die die Reihenfolge der Adressen der Kandidaten-Basic-Blocks in M_i angibt. Dazu werden den Adressen der einzelnen Spaltenvektoren ihre Zeilennummern zugeordnet, und die so entstandenen Paare werden in einer $c \times d$ -Matrix $W = (w_{i,j})$ gespeichert. Ein Eintrag $w_{i,j}$ besteht aus dem Paar $(j, b_{i,j})$. Die Paare jeder Spalte werden anschließend nach aufsteigenden Adressen sortiert. Die Nummern aus den Paaren der i -ten Spalte bilden dann – hintereinander geschrieben – die Elemente der c -stelligen Permutation π_i . Nach der Sortierung lässt sich ein Eintrag $w_{i,j}$ durch das Paar $(\pi_i(j), b_{i,j})$ beschreiben.
5. ClamAV-Signaturen erlauben ausschließlich die Suche nach Byte-Sequenzen in einer festen Reihenfolge, daher wird nun nach einer Teilsequenz von Kandidaten-Basic-Blocks gesucht, die in aufsteigender Reihenfolge ihrer Startadressen in allen Schadprogrammen vorkommen. Fasst man die Spaltenvektoren von W als Sequenzen auf, entsteht eine solche Teilsequenz durch Anwendung eines Algorithmus für das *k-LCS*-Problem auf

den Nummern der einzelnen Paare. Das Ergebnis der Ausführung dieses Schritts ist eine Sequenz $K = k_1 \dots k_\ell$, die eine längste gemeinsame Teilsequenz aller π_i ist.

6. Da die Länge der im vorherigen Schritt berechneten gemeinsamen Teilsequenz K in der Praxis sehr groß sein kann (Schadprogramme, die mehrere Tausend strukturell identische Basic-Blocks gemeinsam haben, sind keine Seltenheit), wird nun die Länge von K durch Streichung von Elementen nach oben begrenzt. Die gekürzte Sequenz wird mit K' bezeichnet, die resultierende Länge mit ℓ' . Dieser Schritt macht die erzeugte Signatur „generischer“.
7. Es werden aus allen Spalten der Matrix W die Paare gestrichen, deren zugeordnete Nummer nicht in der Sequenz K' vorkommt. Die resultierende $\ell' \times d$ -Matrix wird mit $W' = (w'_{i,m})$ bezeichnet.
8. Die Matrix W' wird spalten- und zeilenweise durchlaufen und eine $\ell' \times d$ -Matrix $I_W = (i_{W_{i,m}})$ mit den Byte-Sequenzen der Instruktionen der Basic-Blocks aus den Elementen von W' angelegt.
9. Die einzelnen Zeilen von I_W stellen eine Menge von Byte-Sequenzen dar, auf welche die in Abschnitt 3.5 beschriebene Heuristik für das k -LCS-Problem angewendet wird. Die gemeinsamen Teilsequenzen, die sich aus den einzelnen Zeilen ergeben, werden in einem Zeilenvektor T der Länge ℓ' gespeichert.
10. Der Vektor T und die Matrix I_W werden gleichzeitig zeilenweise durchlaufen, dabei wird für jede gemeinsame Teilsequenz t_m ein regulärer Ausdruck r_m erzeugt, der die Byte-Sequenzen der Zeilen von I_W akzeptiert. Dabei wird jeweils nach zusammenhängenden Teilstrings in den berechneten gemeinsamen Teilsequenzen gesucht und diese hintereinandergeschrieben und mit einem Wildcard-Symbol $\lceil * \rceil$ voneinander getrennt. Mit den einzelnen r_m wird ebenso verfahren. Dies ergibt einen neuen regulären Ausdruck der Form $R = r_1 \lceil * \rceil \dots \lceil * \rceil r_m$.
11. Nach der Erzeugung eines geeigneten Namens ($\lceil \langle \text{name} \rangle \rceil$) für die Signatur der Malware-Familie ergibt sich eine ClamAV-Signatur

$$S = \lceil \langle \text{name} \rangle : 0 : * : \rceil \text{hex}_R(r_1) \lceil * \rceil \dots \text{hex}_R(r_m).$$

$\text{hex}_R(r)$ bezeichnet dabei die Darstellung des regulären Ausdrucks r nach der Umwandlung der enthaltenen Byte-Sequenzen in hexadezimale Darstellung. Der Teilausdruck $\lceil : 0 : * : \rceil$ gibt an, dass es sich bei der generierten ClamAV-Signatur um eine Signatur im erweiterten Signaturformat handelt, die auf alle Dateien und alle Dateipositionen anzuwenden ist.

3.4 Längste gemeinsame Teilsequenzen von Permutationen

Schritt 5 der Beschreibung aus Abschnitt 3.3 erfordert die Berechnung einer längsten gemeinsamen Teilsequenz (LCS) von k Sequenzen der Länge n . Da dieses Problem für $k > 2$ NP-hart ist, ist im Allgemeinen eine exponentielle Laufzeit $O(n^k)$ für die Berechnung einer exakten Lösung zu erwarten. Da es sich jedoch bei der Eingabe um Sequenzen aus Elementen von n -stelligen Permutationen handelt, lässt sich die Laufzeit wesentlich verbessern: Die Problemgröße wird verringert, indem die zwei – bezüglich ihres Hamming-Abstands – ähnlichsten Sequenzen durch ihre LCS ersetzt werden. Aus den restlichen Sequenzen werden alle Elemente gestrichen, die nicht in der berechneten LCS vorkommen, da sie nicht Teil der LCS aller Sequenzen sein können. Mit dem reduzierten Problem wird so lange rekursiv fortgefahren, bis nur noch zwei Sequenzen übrig sind. Die Berechnung einer LCS von zwei Sequenzen ist zum Beispiel mit dynamischer Programmierung (DYN-LCS) in Polynomialzeit möglich. Ein Algorithmus HAMMINGLCS,

der diese Idee umsetzt, wird in Abbildung 1 im Pseudocode dargestellt. Zur Bestimmung der zwei ähnlichsten Sequenzen wird paarweise der Hamming-Abstand berechnet. Neben der Ersetzung zweier Sequenzen durch ihre LCS wird als zusätzliche Optimierung noch eine Liste von Indizes geführt, deren zugehörige Sequenzen gelöscht werden können, da sie mehrfach in der Eingabe vorkommen. Für die Streichung von Elementen aus Sequenzen wird die Funktion RETAINALPHABET verwendet.

Satz 3.3. *Der Algorithmus HAMMINGLCS berechnet bei Eingabe von k Sequenzen S_i mit $0 < i \leq k$, die aus Elementen von n -stelligen Permutationen bestehen, die längste gemeinsame Teilsequenz aller S_i .*

Beweis. Die Eingabesequenzen haben die Eigenschaft, dass sie alle das gleiche endliche Alphabet verwenden. Da die Sequenzen außerdem aus Elementen von n -stelligen Permutationen bestehen, kommt jedes Element in einer Sequenz genau einmal vor. Der Hamming-Abstand zweier Sequenzen S_i und S_j gibt daher die Anzahl der Fixpunkte der Permutation $\pi = \pi_{S_i} \circ \pi_{S_j}$ an, die sich aus der Hintereinanderausführung der Permutationen der beiden Sequenzen ergibt. Die LCS von S_i und S_j bestimmt ein neues Alphabet Σ_π , das nun auf alle anderen Sequenzen durch Streichung von Elementen, die nicht in Σ_π enthalten sind, aufgeprägt wird. Die Streichung ist korrekt, da die LCS von S_i und S_j in jedem Fall eine Teilsequenz der LCS aller Sequenzen der Eingabe ist. Der Algorithmus terminiert, da jeder rekursive Aufruf die Problemgröße um mindestens Eins verringert und ein Problem der Größe Zwei mit der Methode der dynamischen Programmierung gelöst wird. Nach $k - 1$ Rekursionen bleibt also nur noch die LCS von allen Sequenzen S_i übrig. \square

Eine Laufzeitabschätzung kann durch Aufstellen einer Rekursiven Laufzeitfunktion $T(k, n)$ erfolgen. Für HAMMINGLCS ist somit durch

$$\begin{aligned} T(2, n - k + 1) &= n^2 \\ T(k, n) &= \binom{k}{2} n + n^2 + (k - 1)n(n - 1) + T(k - 1, n - k + 1) \\ &= O(k^4 n + k^2 n^2) \end{aligned}$$

nach oben beschränkt. Diese Laufzeit ist – wie bereits vermutet – signifikant besser als die exponentielle Laufzeit für die Berechnung einer LCS für k Sequenzen im allgemeinen Fall.

3.5 Eine Heuristik für das k -LCS-Problem

Der Algorithmus aus Abbildung 1 lässt sich unverändert auch in Schritt 9 aus Abschnitt 3.3 zur Berechnung von gemeinsamen Teilsequenzen zu verwenden. Die Byte-Sequenzen der Eingabe sind dort jedoch nicht aus Elementen von Permutationen entstanden und haben demnach auch keine besondere Struktur. Jede Sequenz verwendet außerdem eine potentiell andere Teilmenge des Alphabets der durch ein Byte darstellbaren Zeichen. Eine optimale Lösung für das allgemeine k -LCS-Problem ist daher nicht zu erwarten. Der Algorithmus HAMMINGLCS kann jedoch dazu verwendet werden, um bei ansonsten gleicher Zeitkomplexität eine gemeinsame – nicht notwendigerweise längste – Teilsequenz von k Sequenzen zu berechnen. Um eine Heuristik für k -LCS zu erhalten, wird daher bei der Berechnung des Hamming-Abstands bei unterschiedlich langen Sequenzen die Differenz der Längen zum Abstand hinzuaddiert. Nach Konstruktion des Algorithmus ist sichergestellt, dass nur gemeinsame Teilsequenzen berechnet werden. Es kann jedoch vorkommen, dass keine gemeinsame Teilsequenz gefunden wird. In diesem Fall wird die leere Sequenz $\langle \rangle$ zurückgegeben. Nach den obigen Ausführungen kann also als Ergebnis festgehalten werden:

```

RETAINALPHABET( $S = s_1 \dots s_m, \Sigma$ )
1  result ← NIL
2  for  $i \leftarrow 1$  to  $m$ 
3      do if  $s_i \in \Sigma$ 
4          then append(result,  $s_i$ )
5  return result

HAMMINGLCS( $S_1, \dots S_k$ )
1  if  $k = 2$                                 ▷ Löse Problem der Größe 2 mit dynamischer Programmierung
2      then return DYN-LCS( $S_1, S_2$ )
3  kill ← NIL                                  ▷ Liste mit Indizes zu löschender Sequenzen
4   $hmin \leftarrow +\infty, hcur \leftarrow 0$       ▷ Minimaler und aktueller Hamming-Abstand
5   $shd1 \leftarrow 0, shd2 \leftarrow 0$ 
   ▷ Indizes der Sequenzen mit dem kleinsten Hamming-Abstand suchen
6  for  $i \leftarrow 1$  to  $k$ 
7      do for  $j \leftarrow 1$  to  $i$ 
8          do if  $i = j$ 
9              then continue
10              $hcur \leftarrow \Delta_h(S_i, S_j)$           ▷ Hamming-Abstand berechnen
11             if  $hcur = 0$ 
12                 then append(kill,  $i$ )                ▷ Identische Sequenz löschen
13             elseif  $hcur < hmin$ 
14                 then  $hmin \leftarrow hcur$ 
15                  $shd1 \leftarrow i, shd2 \leftarrow j$ 
16 if  $|kill| = k - 1$                                 ▷ Alle Sequenzen bis auf eine identisch?
17     then return  $S_1$                                 ▷ Diese ist dann LCS
   ▷ Berechne LCS der „ähnlichsten Sequenzen“ mit dynamischer Programmierung
18  $minlcs \leftarrow$  DYN-LCS( $S_{shd1}, S_{shd2}$ )
19 sublist ← NIL
20 for  $i \leftarrow 1$  to  $k$ 
21     do if  $i \in kill$  or  $i = shd2$ 
22         then continue
23     elseif  $i = shd1$ 
24         then append(sublist, minlcs)
25     else append(sublist, RETAINALPHABET( $S_i, minlcs$ ))
26 if  $|sublist| = 1$                                 ▷ Nur noch eine Sequenz übrig?
27     then return first(sublist)
28 else return HAMMINGLCS(sublist)                    ▷ Rekursion

```

Abb. 1: Ein Algorithmus zur Berechnung von gemeinsamen Teilsequenzen

Satz 3.4. Der Algorithmus HAMMINGLCS berechnet bei Eingabe von k Sequenzen S_i mit $0 < i \leq k$ und maximaler Länge $n = \max_{0 < i \leq k} |S_i|$ in Zeit $O(k^4n + k^2n^2)$ eine gemeinsame Teilsequenz aller S_i . \square

4 Implementierung

Das vorgestellte Verfahren zur automatisierten Generierung von Erkennungssignaturen wurde in Form des kommandozeilenorientierten Programms *SigGen* in der Programmiersprache Java implementiert. Zur Automatisierung von BinDiff-Aufrufen wurde ein Plugin für den Disassembler IDA Pro in C++ entwickelt, welches die Entwicklung von Java-basierten Plugins und die vollständige Steuerung von IDA ermöglicht.

Ein typischer Arbeitsablauf mit *SigGen* ist der folgende: Zunächst wird, wie bereits beschrieben, eine Menge von Schadprogrammen mittels VxClass in Familien klassifiziert und eine Familie zur Signaturgenerierung ausgewählt. Die IDA-Pro-Datenbanken der einzelnen Malware-Dateien dienen *SigGen* als Eingabe. Es folgt der eigentliche Programmaufruf über die Kommandozeile. Je nach Größe und Komplexität der Malware kann die Generierung einer Familiensignatur einige Minuten dauern. Während der Ausführung von *SigGen* werden die IDA-Pro-Datenbanken geöffnet und in ein XML-Format exportiert. Dieses Format dient als Eingabe für das Programm BinDiff, das so genannte *Difference Reports* ebenfalls in einem XML-Format speichert. Die einzelnen Reports werden eingelesen, und es wird das vorgestellte Verfahren angewendet. Die für die entgültigen Signaturen benötigten Byte-Sequenzen werden mit IDA Pro aus den IDA-Pro-Datenbanken extrahiert.

Die so erzeugte ClamAV-Signatur im erweiterten Signaturformat wird in einer Signaturdatenbank gespeichert und ist damit für den Scanner von ClamAV direkt nutzbar und kann zum Beispiel auf falsche Positive getestet werden oder Grundlage für weitere Experimente bilden.

5 Evaluierung

Abschnitt 3.2 stellt einige Anforderungen an ein Verfahren zur automatisierten Signaturgenerierung. Es ist bereits klar, dass das beschriebene Verfahren vollständig automatisiert abläuft und Signaturen aus den Instruktionsbytes der Malware erstellt. Nach Konstruktion werden immer gültige reguläre Ausdrücke für die Signatur erzeugt, da genau ein Wildcard-Symbol weniger hinzugefügt wird, als zusammenhängende Teilstrings in den gemeinsamen Teilsequenzen der jeweiligen Byte-Sequenzen der Schadprogramme auftreten. Dies sind bei Verwendung des Wildcard-Symbols `[*]` nicht mehr, als nötig.

Im Folgenden werden die von *SigGen* erzeugten Signaturen auf falsche Positive und falsche Negative bei der Erkennung von Malware durch den Scanner ClamAV getestet. Es wurden für zehn Familien von Malware mit jeweils 5 bis 30 Mitgliedern Signaturen erzeugt und in einer ClamAV-Signaturdatenbank gespeichert. Unter falschen Positiven ist in diesem Zusammenhang zu verstehen, dass bei Verwendung der Signaturdatenbank in ClamAV das Vorhandensein eines Schadprogramms gemeldet wird, obwohl es sich bei der betreffenden Datei nicht um Malware handelt. Analog ist unter falschen Negativen das Nicht-Erkennen von vorhandener Malware zu verstehen. Ein Test auf falsche Negative testet also gewissermaßen die Korrektheit der erzeugten Signaturen. Nach Konstruktion der für die ClamAV-Signaturen verwendeten regulären Ausdrücke ist jedoch eine erzeugte Signatur entweder korrekt, oder es wird keine Signatur erzeugt, sodass falsche Negative nur dadurch entstehen können, dass der Scanner von ClamAV getarnte Malware in vielen Fällen nicht entpacken kann. Die in dem Softwarewerkzeug VxClass verwendete generische Entpacker-Komponente kann hingegen eine viel größere Zahl von getarnter Malware erfolgreich entpacken. Da *SigGen* auf Basis der vorklassifizierten – bereits entpackten – Disassembler-Datenbanken arbeitet, können bei der Verwendung der erzeugten Signaturen auf

gepackten Dateien falsche Negative auftreten.

5.0.1 Testverfahren für falsche Positive

Um die Reproduzierbarkeit von Tests zu gewährleisten, wurde das Betriebssystem Microsoft Windows XP Professional in eine virtuelle Maschine auf Basis von VirtualBox [SunM08] installiert. Anschließend wurden alle verfügbaren Updates für das Windows-Betriebssystem eingespielt und einige, für einen typischen „Büroarbeitsplatz“ verwendete Softwarepakete installiert. Bei den in der virtuellen Maschine installierten Programmen handelte es sich um ein Office-Paket, einige Dateibetrachter, Software zur Fernwartung sowie einen Web-Browser. Alle Dateien der virtuellen Maschine wurden anschließend mit dem Scanner ClamAV in Version 0.94 und der erzeugten Signaturdatenbank gescannt. Der Test mit der oben beschriebenen virtuellen Maschine wird durch die System-Scans von drei weiteren Rechengesystemen ergänzt, die sich im produktiven Einsatz befinden:

- „Büro-Server“ – Eine Standard-Installation mit dem Betriebssystem Microsoft Windows 2003 Server SBS. Dieses System wird eingesetzt als Datei- und Mailserver.
- „Web-Host“ – Ein Linux-basierter Web-, FTP-, Mail- und Datenbankserver.
- „Entwickler“ – Windows-basierte Systems, welches zur Entwicklung von SigGen verwendet wurde.

Die Ergebnisse der einzelnen System-Scans wurden jeweils mit einem zweiten System-Scan unter Verwendung der ClamAV-eigenen Signaturdatenbank verglichen. Die gesammelten Daten werden in Tabelle 1 festgehalten.

	Büroarbeitsplatz	Büro-Server	Web-Host	Entwickler
Verzeichnisse	2 137	12 250	25 382	34 057
Dateien	21 315	164 845	329 594	279 053
Gescannte Daten	4,91 GiB	101,58 GiB	58,59 GiB	34,19 GiB
Als infiziert erkannt	0 (8 ^a)	0	4 ^b	0
Als infiz. erkannt (2. Lauf)	0 (8 ^a)	0	4	0

^a Für den Scanner von ClamAV zu große Archivdateien.

^b Tatsächliches Vorkommen eines Schadprogramms aus einer der Familien, für die eine Signatur generiert wurde.

Tab. 1: Ergebnisse von kompletten ClamAV-Scandurchläufen

Bei Betrachtung der Tabelle fällt auf, dass vier von den Dateien des Rechengesystems „Web-Host“ als Schadprogramm erkannt wurden. Der Vergleich mit dem zweiten System-Scan unter Verwendung der mitgelieferten ClamAV-Signaturdatenbank und eine anschließende manuelle Analyse bestätigten das Vorhandensein von vier identischen Exemplaren einer Variante des Trojanischen Pferds „Phänomen Downloader“ in einem Anhang einer gespeicherten Email. Die Listings 2 und 3 zeigen die in der mitgelieferten ClamAV-Signaturdatenbank enthaltene und die von SigGen aus 22 Mitgliedern der Malware-Familie erzeugte Signatur.

Die acht als infiziert erkannten Dateien aus den Dateien der virtuellen Maschine „Büroarbeitsplatz“ sind Dateiarhive, die Dateien enthalten, deren Kompressionsfaktor ungewöhnlich groß ist. Dies ist eine dokumentierte Funktion des ClamAV-Scanners [Clam07], die sich konfigurieren lässt. Zusammengefasst finden sich in den Einträgen der Tabelle also keine Anhaltspunkte für falsche Positive.

```

1 Trojan.OnLineGames-65:1:*:47616d657320446f777e6c6f6164657200000000446f777e6
2 c6f6164696e672e2e2e00005061757365642e2e2e00000025692525206f6620252e31666d62
3 20646f777e6c6f616465642028252e31666d6229000025693a253032692052656d61

```

Listing 2: Signatur aus ClamAV-Signaturdatenbank für eine Variante des Trojanischen Pferds „Fenomen Downloader“

```

1 Worm.SigGen-20080528144600-1631:0:*:5c24*c64424*0000*048d*6860*4100*c64424*
2 5803e8*0883c40c8d5424*4100*3bc3*0f86*0000*83ec0c8d*8bcc89*c68424b0020000*83
3 ec0c8d*85c974*c74614000000*8b4c2408*8b4610

```

Listing 3: Von SigGen erzeugte Signatur für 22 Mitglieder der Familie „Fenomen Downloader“

5.1 Übertragbarkeit

Um die Übertragbarkeit der für eine Malware-Familie erzeugten Signaturen auf noch unbekannte Mitglieder dieser Familie empirisch nachzuweisen, wurden aus zwei Familien mit je 22 und 30 Mitgliedern jeweils vier Schadprogramme ausgewählt und mit SigGen Signaturen für die Auswahlen erzeugt. Anschließend wurden die beiden erzeugten Signaturen mit ClamAV auf den ausführbaren Dateien der jeweiligen Familie getestet. Das Ergebnis dieses Experiments zeigt Tabelle 2. Alle Schadprogramme der Familie wurden erkannt. Ein Scan des oben erwähnten

	Auswahl 1	Auswahl 2
Gescannte Dateien	22	30
Gescannte Daten	2,75 MiB	4,25 MiB
Als infiziert erkannt	22	30
Signaturlänge	1 876	2 065

Tab. 2: Anwendung einer erzeugten Signatur für vier Schadprogramme auf die gesamte Familie

Rechensystems „Entwickler“ führte zu keinen falschen Positiven. Basierend auf der sehr kleinen Testmenge scheinen sich die von SigGen erzeugten Signaturen für die Erkennung noch unbekannter Mitglieder einer Malware-Familie zu eignen.

5.2 Performanz des Verfahrens

Die Laufzeiten für die Erzeugung von Signaturen für zehn Malware-Familien mit jeweils 5 bis 30 Mitgliedern und die bearbeiteten Datenmengen werden in Tabelle 3 gezeigt. Alle Signaturen wurden auf einem System mit einem Intel Core2-Duo-Prozessor mit 2,13 GHz und 3,0 GiB Hauptspeicher erzeugt. Die IDA-Pro-Datenbanken und die BinDiff-Ausgaben wurden auf einem eigenen Datenträger gespeichert. Die angegebenen Laufzeit-Kennzahlen scheinen die erhoffte praktische Einsetzbarkeit des Verfahrens auf modernen Rechensystemen zu bestätigen.

6 Fazit

Das vorgestellte Verfahren zur automatisierten Generierung von Signaturen für Stämme von Malware und dessen prototypische Implementierung erfüllen die in 3.2 gestellten Ziele. Die vollständige Automatisierung des Verfahrens erlaubt eine Verkürzung der Zeitspanne von der

Familie	1	2	3	4	5	6	7	8	9	10
Mitglieder	14	14	5	10	5	30	5	12	22	5
Laufzeit (s)	66,1	162,5	22,4	168,6	49,2	484,5	23,7	197,2	365,3	208,4
Signatur-Länge	1 189	974	786	1 458	1 262	1 327	1 766	871	1 631	1 216

Tab. 3: Laufzeiten bei der Erzeugung von Signaturen für zehn Malware-Familien

Entdeckung eines neuen Schadprogramms bis zur Verteilung einer Signatur für die neu entdeckte Malware, da die zeitaufwändige und fehleranfällige manuelle Analyse entfällt.

Da die zur Evaluation von SigGen verwendete Testmenge sehr klein ist, lassen sich keine sicheren Schlüsse im Hinblick auf die Übertragbarkeit auf die Erkennung noch unbekannter Mitglieder einer Malware-Familie ziehen. Hier ist eine genauere Untersuchung der Vorhersagekraft der erzeugten Signatur erforderlich.

Die starke Abhängigkeit von SigGen von den verwendeten Softwarewerkzeugen BinDiff und VxClass lässt eine zukünftige Integration von SigGen als Komponente von VxClass sinnvoll erscheinen.

Literatur

- [Ayco06] J. Aycock: Computer Viruses and Malware. Advances in Information Security, Springer, Verlin, 1st edition Aufl. (2006).
- [Clam07] ClamAV-Projekt: ClamAV Official FAQ (2007), <http://www.clamav.org/support/faq/>, [Online; Stand 08. März 2009].
- [Clam08] ClamAV-Projekt: About ClamAV (2008), <http://www.clamav.net/about/>, [Online; Stand 08. März 2009].
- [DuRo05] T. Dullien, R. Rolles: Graph-Based Comparison of Executable Objects. In: *SSTIC '05, Symposium sur la Sécurité des Technologies de l'Information et des Communications* (2005).
- [Flak04] H. Flake: Structural Comparison of Executable Objects. In: U. Flegel, M. Meier (Hrsg.), *DIMVA, GI* (2004), *LNI*, Bd. 46, 161–173.
- [GmbHa] zynamics GmbH: VxClass — Automatic classification of malware and trojans into „families“. <http://www.vxclass.com/>, [Online; Stand 08. März 2009].
- [GmbHb] zynamics GmbH: zynamics BinDiff. <http://www.zynamics.com/bindiff.html>, [Online; Stand 08. März 2009].
- [GrTe07] D. Gryaznov, J. Telafici: What a waste — The anti-virus industry DOS-ing itself. Tech. Rep., McAfee Avert Labs, USA (2007).
- [Kojm09] T. Kojm: Creating signatures for ClamAV (2009), <http://www.clamav.net/doc/latest/signatures.pdf>, [Online; Stand 08. März 2009].
- [Maie78] D. Maier: The Complexity of Some Problems on Subsequences and Supersequences. In: *J. ACM*, 25, 2 (1978), 322–336.
- [SA08] H.-R. SA: The IDA Pro Disassembler and Debugger (2008), <http://www.hex-rays.com/idadpro/>, [Online; Stand 08. März 2009].

-
- [SunM08] I. Sun Microsystems: VirtualBox – professional, flexible, open (2008), <http://www.virtualbox.org/wiki/VirtualBox>, [Online; Stand 15. Mai 2008].
- [SzFe01] P. Ször, P. Ferrie: Hunting for metamorphic. In: *Virus Bulletin International Conference (VB2001)*, Virus Bulletin (2001), 123–144.