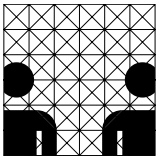
Universität Hamburg
Fachbereich Informatik



Diplomarbeit

Entwurf einer Sprache zur Klassifizierung von Malware-Vorfällen

Michel Messerschmidt

2. April 2004

Erstbetreuer: Prof. Dr. Klaus Brunnstein
Zweitbetreuer: Dr. Hans-Joachim Mück

Danksagung

Hiermit möchte ich mich bei Birte Schmude für die große Unterstützung in den letzten Monaten bedanken, die es mir erst ermöglichte, mich ganz auf diese Arbeit konzentrieren zu können.

Sibilla und Wolfgang Schmude danke ich für das ausführliche Korrekturlesen.

Und schließlich danke ich Herrn Prof. Brunnstein und Herrn Dr. Mück für die Betreuung dieser Arbeit.

Inhaltsverzeichnis

1 Überblick

In dieser Diplomarbeit wird versucht, ein allgemein verwendbares Datenformat (bzw. eine Auszeichnungssprache) zu entwerfen. Ziel dieser Sprache ist die einheitliche und umfassende Beschreibung von Malware-Vorfällen.

Der Ausgangspunkt sind dabei die Logdatei-Formate von Anti-Viren-Produkten sowie das im antiVirusTestCenter der Universität Hamburg verwendete Logdatei-Format. Die zu entwickelnde Sprache soll dabei aber nicht nur den Anforderungen des antiVirusTest-Centers genügen, sondern sich auch für weitere Anwendungen einsetzen lassen.

In Kapitel 2 wird der aktuelle Zustand der Testauswertung im antiVirusTestCenter beschrieben sowie die Motivation für diese Arbeit dargelegt. Daraufhin werden in Kapitel 3 weitere Anwendungsgebiete aufgezeigt, die von einem allgemein verwendbaren Format profitieren könnten. Kapitel 4 stellt das Konzept für die zu entwerfende Sprache vor. Nachdem in Kapitel 5 die technischen Grundlagen erläutert worden sind, wird die Umsetzung des Konzepts in Kapitel 6 beschrieben. In Kapitel 7 wird ein abschließender Vergleich der Umsetzung mit den Zielen durchgeführt.

2 Einführung

2.1 Motivation

Das antiVirusTestCenter (aVTC) der Universität Hamburg führt seit Jahren Tests von Antiviren-Produkten durch, um deren Erkennungsraten zu ermitteln. Dazu werden die in den Testläufen von den Anti-Viren-Produkten erzeugten Logdateien analysiert und daraus die Ergebnisse ermittelt[1].

Da jeder Hersteller eines Anti-Viren-Produkts ein anderes Logdatei-Format verwendet, werden diese im Rahmen der Testauswertung in ein einheitliches Datenformat umgewandelt, bevor die weitere Auswertung erfolgt. Anfangs wurden dazu noch für jedes Produkt eigene Skripte verwendet. Aber mit zunehmendem Umfang der Tests und der Einführung neuer, spezieller Testmethoden stieg der Aufwand für die Auswertung zunehmend. Daher wurden neue Auswertungsmethoden entwickelt, um den Aufwand zu reduzieren. Insbesondere wurden allgemein verwendbare Auswertungsabläufe identifiziert und daraus ein generisches Auswertungsprogramm erstellt, so dass die produkt-spezifischen Anteile getrennt gewartet werden konnten. Dazu wurde natürlich ein generisches Datenformat benötigt.

Leider ist das bisher verwendete Datenformat nicht sehr gründlich entworfen worden, sondern wurde aus Zeitgründen aus den bisherigen Skripten abgeleitet. Somit ist es für die aktuellen Tests nicht mehr geeignet und führt zu einer deutlich aufwändigeren Testauswertung als es notwendig wäre. Daher besteht im aVTC der Bedarf nach einem verbesserten Datenformat für die Klassifizierung der Testdateien.

Darüber hinaus ist zu beobachten, dass Klassifikationen von Anti-Viren-Produkten oft wenig differenziert sind. Zum Teil ist der Logdatei schon nicht mehr zu entnehmen, ob eine Datei nicht gescannt werden konnte oder untersucht und für nicht infiziert befunden wurde. Ein sorgfältig entworfenes Datenformat sollte auch solche Umstände berücksichtigen. Eine Verbesserung kann in dieser Hinsicht natürlich nur erreicht werden, wenn Anti-Viren-Hersteller dieses Datenformat auch in ihren Produkten verwenden würden.

Eine andere Beobachtung bezieht sich auf die Verständlichkeit und Austauschbarkeit von Klassifikationsdaten. Zum einen sind viele Meldungen von Anti-Viren-Produkten keines-

[1] Für eine detaillierte Beschreibung der Testabläufe sei auf [Seedorf 2002] und [Messerschmidt 2002] verwiesen.

wegs eindeutig interpretierbar[2]. Hier sollte ein neu entworfenes Datenformat eindeutige Aussagen hinsichtlich des Sicherheitsrisikos erfordern. Zum anderen stellen Produkte ihre Klassifikationsdaten zum Teil nur über eigene Anzeigeprogramme zur Verfügung. Gerade im Zusammenhang mit den immer komplexeren Sicherheitsarchitekturen heutiger Computersysteme und -netzwerke könnte ein einheitliches Datenformat die Verwaltung und Übersichtlichkeit sicherheitsrelevanter Informationen deutlich erhöhen (unter anderem durch die Verwendung unabhängiger Analyse-Anwendungen). Dabei sollten idealerweise nicht nur Anti-Viren-Produkte sondern auch andere Sicherheitstechnologien berücksichtigt werden.

2.2 Grundlegende Definitionen

In diesem Abschnitt werden einige in dieser Arbeit verwendete Begriffe definiert.

Dateneinheit wird in dieser Arbeit als Bezeichnung für eine möglichst atomare Unterteilung von Daten verwendet. Diese können je nach betrachtetem System zum Beispiel Bits, Bytes, 32Bit-Worte, ... sein.

Datenobjekt Eine Menge von Dateneinheiten, die sich durch einen logischen Zusammenhang ihrer Inhalte von anderen Dateneinheiten abgrenzt. Ein logischer Zusammenhang von Daten kann etwa durch ein maschinen-interpretierbares Programm („Software“) oder eine Programm-Funktion, aber auch durch einen für Menschen ersichtlichen Zusammenhang (wie zum Beispiel einen Text oder eine Grafik) gegeben sein.

Datei ist eine durch ein Dateisystem verwaltete Unterteilung von Dateneinheiten. Ein Dateisystem dient einerseits der Abstraktion von der physikalischen Datenspeicherung und andererseits zur Verwaltung und Organisation von Dateneinheiten (ähnlich einer Datenbank). Eine Datei ist dabei eine als Einheit adressierbare Menge von Dateneinheiten, die durch Attribute wie Name, Pfad, Größe, Zeitstempel und Zugriffsrechte gekennzeichnet ist. Durch eine Datei ist nicht notwendigerweise ein logischer Zusammenhang der Datei-Inhalte begründet. Eine Datei kann zwar mit einem Daten-Objekt identisch sein, aber auch mehrere Daten-Objekte oder nur einen Teil eines Daten-Objekts beinhalten. Daten-Objekte können auch unabhängig von einem Dateisystem existieren (zum Beispiel Bootsektoren). Ein universell entworfenes Datenformat kann also nicht auf Dateien als atomaren Objekten basieren.

Dokument Im Zusammenhang mit Auszeichnungssprachen (wie zum Beispiel XML) werden Dateien manchmal auch als Dokumente bezeichnet.

[2] Beispiele: „contains macros“, „could be a corrupted executable file“ oder „Found application Tool/W311“

Malware ist die Kurzform von „malicious software“ und somit ein Oberbegriff für jegliche Art Software, die man als „bösartig“ oder „schädlich“ einschätzen würde. Deshalb lässt sich die Entscheidung, ob Software Malware ist oder nicht, leider nicht eindeutig treffen; manchmal ist dies sogar vom Anwendungskontext abhängig. Dementsprechend gibt es durchaus verschiedene Definitionen von „Malware“. In dieser Arbeit wird folgende Definition von [Brunnstein 1999] verwendet:

> A software or module is called „malicious“ („malware“), if it is *intentionally dysfunctional*, and if there is sufficient evidence (e.g. by observation of behaviour at execution time) that dysfunctions may adversely influence the usage or the behaviour of the original software.

Malware wird meist nach Arten unterteilt (also z. B. Viren, Trojaner, Hoaxes) und nach mehreren weiteren Kriterien klassifiziert (z. B. ob die Malware auf ein vernetztes System angewiesen ist). Dies wird in Abschnitt 3.1.2 näher beschrieben.

Oft wird Malware dabei in die Kategorien „viral“ (Viren und Würmer) und „nichtviral“ (alles andere) unterteilt.

Virus Als (Computer-)Virus wird jene Malware bezeichnet, die die Fähigkeit zur Selbst-Replikation auf einem einzelnen System besitzt.

Es existieren verschiedene Definitionen für Viren, unter anderem auch eine formale Definition [Cohen 1984] (ein Überblick findet sich in [Bontchev 1998, p. 23ff]). In dieser Arbeit wird folgende Definition von [Brunnstein 1999] verwendet:

> Any software that reproduces (or „self-replicates“), possibly depending on specific conditions, at least in 2 consecutive steps upon at least one module each (called „host“) on a single system on a given platform, is called a „*virus*“ (for that platform). A viral host may be compiled (e.g. boot and file virus) or interpreted (e.g. script virus).

Daher sind Viren (und Würmer, s. u.) die einzige Malware-Art, für die eindeutig entschieden werden kann, ob eine gegebene Software in diese Kategorie fällt oder nicht.

Wurm Als (Computer-)Wurm wird jene Malware bezeichnet, die die Fähigkeit zur Selbst-Replikation auf vernetzten bzw. verteilten Systemen besitzt.

Diese werden in [Brunnstein 1999] analog zu Viren definiert:

> Any software that reproduces (or „propagates“), possibly depending on specific conditions, in at least two parts (nodes) of a connected system on a given platform, is called a (platform) „*worm*“, if it has the ability to communicate with other instances of itself, and if it is able to transfer itself to other parts of the network using the same platform.

Malware-Vorfall („malware incident“) ist die Bezeichnung für das Auftreten bzw. Aktivieren von Malware auf einem Computersystem. Dies beinhaltet sowohl die (aus Malware-Sicht) erfolgreiche Ausführung (unabhängig vom dadurch verursachten Schaden) wie auch fehlgeschlagene oder verhinderte Aktivierungsversuche. Ist Malware auf einem Computersystem lediglich gespeichert, ohne jemals aktiviert zu werden, ist dies kein Malware-Vorfall[3].

Anti-Virus-Produkt Dies ist eine Software, die die Fähigkeit besitzt, virale Software zu erkennen. Außerdem sollte die Aktivierung und Verbreitung von Viren verhindert und bereits infizierte Software wieder desinfiziert werden können.

Anti-Malware-Produkt Diese Produkte haben im Gegensatz zu Anti-Viren-Produkten die Aufgabe nicht nur Viren, sondern jegliche Art von Malware erkennen zu können. Dies ist zwar unmöglich, da ein eindeutiges Unterscheidungsmerkmal fehlt, bekannte Malware sowie deren zugrunde liegenden Mechanismen können aber durchaus erkannt werden. Während es noch vor einigen Jahren ausschließlich Anti-Viren-Produkte gab, wurden im Laufe der Zeit zunehmend Funktionen zur Erkennung nicht-viraler Malware ergänzt, so dass sich viele Anti-Virus-Produkte zu Anti-Malware-Produkten weiterentwickelt haben.

False positive Bezeichnung für die irrtümliche Erkennung eines nicht-viralen Datenobjekts als viral durch ein Anti-Virus-Produkt. Das Vorkommen von „false positives“ lässt sich leider nicht verhindern, da es nicht möglich ist, jegliche existierende Software auf Übereinstimmung mit den Erkennungskriterien von Anti-Viren-Produkten zu testen. Da der Aufwand zur Unterscheidung zwischen Virus-Vorfall und „false positive“ im Einzelfall durchaus recht hoch sein kann, kann ein „false positive“-Vorfall zum Teil genauso ärgerlich sein wie ein Virus-Vorfall. Analog existiert auch der Begriff des „false negative“, also der Nichterkennung viraler Software.

2.3 Überblick über die aVTC-Testauswertung

Dieser Abschnitt beschreibt, wie im antiVirusTestCenter aus einer von einem Anti-Malware-Produkt erzeugten Logdatei Ergebnisse in Form von Malware-Erkennungsraten gewonnen werden. Dabei wird aufgezeigt, an welchen Stellen die bisherigen Datenformate unzureichend sind.

Zuerst wird die originale Logdatei mittels eines Perl-Skripts in ein einheitliches Datenformat konvertiert. Da fast jedes Produkt ein anderes Datenformat verwendet und sich

[3] Je nachdem, wie die Malware auf das System gelangt ist, kann dieser Vorgang allerdings als Malware-Vorfall bezeichnet werden.

dies häufig ändert, ist die Anpassung dieser Skripte der zeitaufwändigste Teil der Testauswertung. Zurzeit wird die originale Logdatei bei der Konvertierung gleich in mehrere neue Dateien aufgespalten:

- eine Datei enthält alle Samples, die als infiziert gemeldet wurden
- eine Datei enthält alle Samples, die als nicht infiziert gemeldet wurden
- alle anderen Zeilen der originalen Logdatei werden in einer weiteren Datei gesammelt

Diese Lösung hat allerdings einige Nachteile. Insbesondere ist die Klassifizierung der Samples auf die Kategorien „infiziert“ und „nicht infiziert“ beschränkt. Es ist aber angedacht, die Datei-Konvertierung und die inhaltliche Aufteilung zukünftig separat durchzuführen.

In den weiteren Auswertungsschritten werden die infizierten Samples sortiert, auf Gültigkeit überprüft[4] und gezählt (wobei mehrfache Meldungen desselben Samples natürlich nicht berücksichtigt werden). Zusätzlich werden die gemeldeten Malware-Namen analysiert und eventuell als „unzuverlässige Erkennung“[5] gewertet.

2.3.1 Relevante Daten

Für die Testauswertung werden folgende Angaben pro Sample benötigt:

- Dateiname
- Absoluter Pfad zur Datei (wird demnächst eventuell nicht mehr benötigt)
- eine Ergebnis-Meldung, ob das Sample als infiziert erkannt wurde oder nicht
- gegebenenfalls eine Malware-Identifikation

Diese Angaben müssen also aus dem Produkt-spezifischen Datenformat extrahiert und anschließend in einem einheitlichen Format wieder gespeichert werden.

2.4 Das Datenformat der aVTC-Testauswertung

Im Prinzip ist dies das einfachste hierfür vorstellbare Format:

- es handelt sich um ein reines Textformat
- es wird der ASCII-Zeichensatz verwendet

[4] Dabei wird im Wesentlichen überprüft, ob die gemeldeten Samples tatsächlich im Testbed enthalten sind.
[5] engl. „unreliable detection“: Alle Meldungen eines Produkts, die ein Malware-Sample nicht exakt identifizieren. Dabei könnte es sich zum Beispiel um eine heuristische Erkennung handeln.

- jede Zeile beschreibt genau eine Datei
- jede Zeile enthält am Anfang den Pfad und Dateinamen, dann genau ein Leerzeichen und die Malware-Identifikation
- Datei- und Pfadnamen müssen DOS-Konventionen entsprechen (also insbesondere nur 8+3 Zeichen lang sein)
- zusätzlich dürfen in der gesamten Datei keine Kleinbuchstaben enthalten sein (diese müssen gegebenenfalls in Großbuchstaben umgewandelt werden)

Beispiel:[6]

```
I:\FILE\23W\MRONON\B\MSUB\5441\EXA_012_.EXE W95/SPACES.1445.B
I:\FILE\23W\MRONON\B\TSAEB\27414\A\EXA_000_.EXE W32/BEAST
I:\FILE\23W\MRONON\C\HIC\3001\A\EXA_000_.EXE W95/CIH.A
```

Genau genommen gibt es noch ein zweites Datenformat, da die Inhalte der originalen Logdatei bei der Konvertierung gleich in infizierte und nicht infizierte Samples getrennt werden. Für die nicht infizierten Samples wird dabei in den weiteren Auswertungsschritten angenommen, dass in jeder Zeile ausschließlich Pfad und Dateiname enthalten sind (da eine Malware-Identifikation ja nicht existiert).

Das hier beschriebene Datenformat wird bereits seit etlichen Jahren verwendet. Über die damaligen Entwurfsprinzipien kann inzwischen nur noch spekuliert werden. Vermutlich sollte es aber nur einfach zu lesen und zu verarbeiten sein und die Konvertierung von den produkt-spezifischen Datenformaten ohne zusätzliche Komplexität ermöglichen. Zusätzlich ist zu bedenken, dass viele der in der Zwischenzeit entwickelten speziellen Testmethoden (wie z.B. die Heureka-Tests oder der Pack-Test) damals noch nicht existierten.

Mit der Einführung zusätzlicher Testbeds, die Samples in komprimierten Archiven enthalten, wurde dieses Datenformat semantisch dahingehend verändert, dass jede Zeile entweder einen Dateinamen enthält oder (falls es sich um ein Archiv handelt) einen Dateinamen, dem ein „\“ und der Name der im Archiv enthaltenen Datei angehängt wird. Diese vermutlich sehr eilig entwickelte Erweiterung der Testauswertung sorgt heute noch für zusätzliche Probleme, da Anti-Viren-Produkte dazu neigen, alle möglichen Dateien als Archiv anzusehen und die Entscheidung zwischen erwünschten und unerwünschten „Anhängseln“ oft nicht einfach zu treffen ist. Beispiel:

[6] Das Einfügen zusätzlicher Zeilenumbrüche in überlange Beispiele wird durch das Zeichen ♪ gekennzeichnet.

```
N:\FILE\23W\MRONON\B\MSUB\5441\ARJ.ARJ\EXA_012_.EXE WIN95.S ⤸
PACES.1445
N:\FILE\23W\MRONON\B\TSAEB\27414\A\ARJ.ARJ\EXA_000_.EXE WIN ⤸
95.BEAST.A
N:\FILE\23W\MRONON\C\HIC\3001\A\ARJ.ARJ\EXA_000_.EXE WIN95. ⤸
CIH.GEN
N:\FILE\23W\MRONON\B\MSUB\5441\ZIP.ZIP\EXA_012_.EXE WIN95.S ⤸
PACES.1445
N:\FILE\23W\MRONON\B\TSAEB\27414\A\ZIP.ZIP\EXA_000_.EXE WIN ⤸
95.BEAST.A
N:\FILE\23W\MRONON\C\HIC\9101\A\ZIP.ZIP\EXA_000_.EXE WIN95. ⤸
CIH.GEN
```

Aus der Einfachheit des Datenformats ergeben sich aber auch entscheidende Beschränkungen:

- Jedes Datenobjekt muss durch eine Datei repräsentiert werden
- Pfad- und Dateiangaben anderer Betriebssysteme können nicht beschrieben werden[7]
- Datei-Inhalte können nur sehr umständlich (und fehleranfällig) dargestellt werden
- Der ASCII-Zeichensatz ist heutzutage nicht mehr ausreichend. Auch wenn es für die aVTC-Tests mittels geeigneter Skripte möglich ist, alle Dateinamen an die DOS-Konventionen anzupassen, ist dies ein zusätzlicher Aufwand und erst recht keine allgemein verwendbare Lösung. Es ist auch nicht garantiert, dass sich alle Malware-Namen mit dem ASCII-Zeichensatz darstellen lassen.

2.5 Datenformate von Anti-Malware-Produkten

Obwohl einige produkt-spezifische Datenformate auch auf ganz anderen Formaten[8] basieren, benutzen viele Produkte ein einfaches textbasiertes Format (wobei Ähnlichkeiten zu dem im aVTC verwendeten Format – aus nahe liegenden Gründen – unverkennbar sind). Meist beschreibt dabei jede Zeile genau eine gescannte Datei. Die Zeile beginnt mit dem Dateinamen (inkl. Pfad) gefolgt von der Ergebnismeldung und gegebenenfalls noch der Malware-Identifikation. Diese Komponenten werden dabei durch (ein oder mehrere) Leerzeichen getrennt. Beispiel:

[7] Für die im aVTC durchgeführten Tests unter Linux werden alle Pfadangaben in den Logdateien vor der Auswertung in die entsprechenden DOS-Pfade umgewandelt. Dieses Verfahren ist jedoch sehr umständlich.

[8] Wie zum Beispiel dBase-Datenbanken, CSV-Tabellen („Comma Separated Value“), RTF-Texte („Rich Text Format“) oder auch HTML-ähnliche Formate.

```
I:\W97M\MELISSA\A@MM\W97_000_.DOC Virus found W97M/Melissa
```

Obwohl dieses Format für die meisten Fälle ausreicht, ist es nicht universell einsetzbar:

(i) Pfade oder Dateinamen, die Leerzeichen enthalten, können nicht beschrieben werden

Manche Produkte sind deshalb dazu übergegangen, den Dateinamen in Anführungszeichen oder andere vermeintlich eindeutige Zeichen zu setzen. Beispiel[9]:

```
[DOS][I:\O97M\TRISTATE\C_DOC\2783TB.DOC] is infected by vir
us [O97M/Tristate.C]
```

Da theoretisch aber jedes Zeichen in einem Dateinamen vorkommen kann[10], ist dies natürlich keine Lösung des Problems. Theoretisch ist es zwar möglich, mittels geeigneter Kodierungen eine universelle Darstellung zu erreichen, allerdings ist dies bei keinem mir bekannten Produkt korrekt implementiert. Somit ergeben sich (wenn auch selten) Probleme, wie in dem folgenden Beispiel[11]. Es handelt sich hierbei um einen Scan der Datei `EXA_000_.EXE`:

```
Objekt: "html" in Pfad "W:\TIKNOCV\MRONON\V\SBV\TPIRCSNU.R\
2NEG\EXA_000_.EXE/[From:  "KHJ" <khj@yahoo.com>][Date:  Fri
, 9 Feb 2001 01:45:00 +0700]". Status: "Virus (erkannt); ar
chiviert in Datei "W:\TIKNOCV\MRONON\V\SBV\TPIRCSNU.R\2NEG\
EXA_000_.EXE"".
```

Die Anführungszeichen werden hier mit verschiedenen Bedeutungen gleichzeitig verwendet und verwirren somit eher, statt die Daten zu strukturieren.

(ii) Datei-Inhalte (wie z.B. bei Zip-Archiven) können nicht eindeutig gemeldet werden

Hier ergibt sich das Problem durch die Tatsache, dass eine Datei durchaus mehrere zu untersuchende Daten-Objekte enthalten kann. Die meisten Produkte lösen dieses Problem, indem die Archiv-Datei als Verzeichnis eines (virtuellen) Dateisystems betrachtet wird. Die enthaltenen Daten-Objekte werden entsprechend als Dateien in diesem Verzeichnis behandelt.

[9] Diese Ausgabe stammt von InoculateIT unter Windows 2000. Die Angabe 'DOS' ist also eher irreführend.
[10] Dies gilt zwar nicht für jedes Betriebssystem, aber für ein allgemein verwendbares Datenformat müssen natürlich alle Betriebssysteme betrachtet werden.
[11] Die originale Zeile wird gekürzt wiedergegeben.

Einige Produkte beschreiben die Archiv-Inhalte, indem sie die Namen der enthaltenen Daten-Objekte mit einem Pfad-Separator an den Archiv-Dateinamen anhängen. Beispiel:

```
N:\FILE\23W\MRONON\C\HIC\3001\A\ZIP.ZIP\EXA_011_.EXE
```

Andere Produkte versuchen, den Archiv-Dateinamen und die Datei-Inhalte durch spezielle Zeichenfolgen logisch zu trennen. Beispiele:

```
N:\FILE\23W\MRONON\C\HIC\3001\A\ZIP.ZIP->FILE/23W/MRONON/C/ ⤶
HIC/3001/A/EXA_001_.EXE
```

```
N:\FILE\23W\MRONON\C\HIC\3001\A\ZIP.ZIP:\FILE\23W\MRONON\C\ ⤶
HIC\3001\A\EXA_001_.EXE
```

```
N:\FILE\23W\MRONON\C\HIC\3001\A\ZIP.ZIP : FILE/23W/MRONON/C ⤶
/HIC/3001/A/EXA_001_.EXE
```

Bei keiner dieser Lösungen ist es jedoch möglich, eindeutig zu unterscheiden, ob es sich um einen Dateinamen oder um eine zusammengesetzte Angabe handelt (auch wenn die Wahrscheinlichkeit einer entsprechend benannten Datei zum Teil sehr gering ist). Zusätzlich sind die entsprechenden Umsetzungen oft schlecht implementiert und sorgen so für zusätzliche Probleme. Wie man sieht werden teilweise in der Archivdatei vorhandene Verzeichnisstrukturen weggelassen, wodurch natürlich Probleme mit identischen Dateinamen entstehen können:

```
P:\ZIP.ZIP\EXA_000_.EXE ... Found the W32/Hybris.gen@MM virus
P:\ZIP.ZIP\EXA_001_.EXE ... Found the W32/Hybris.gen@MM virus
P:\ZIP.ZIP\EXA_002_.EXE ... Found the W32/Hybris.gen@MM virus
P:\ZIP.ZIP\EXA_000_.EXE ... Found the W32/MsInit.worm.a virus
```

Ebenso ist die Ausgabe der meisten Produkte auf eine maximale Pfadlänge beschränkt, so dass einige Ausgaben ihren Sinn komplett verlieren. Beispiel:

```
R:\FILE\23W\MRONON\C\HIC\3001\A\CAB.CAB>>...>>...>>...>>... ⤶
>>...>>...>>...>>...>>EXA_000_.EXE
R:\MALW\MROW\MRONON\W\23W\TINISM\MROW\B\ZIP.ZIP>>...>>...>> ⤶
...>>...>>...>>...>>...>>...>>MALW/MRO...
```

Außerdem ist nicht jeder Archiv-Inhalt eine Datei (z.B in OLE-Archiven) und besitzt einen Dateinamen. Die folgenden Beispiele zeigen, dass die Lösungen hier sehr willkürlich gewählt (und in der Regel nicht dokumentiert) sind:

```
o:\TROJAN\VBS\SEEKER\H\VBS_000_.HTM->(SCRIPT1)->(SCRENC)    ⤸
Infected: JS/Seeker.E*
```

```
O:\TROJAN\VBS\SEEKER\H\VBS_000_.HTM ... is OK.
O:\TROJAN\VBS\SEEKER\H\VBS_000_.HTM\00000089.js\00000089.js ⤸
 ... Found the JS/Seeker.gen.h trojan !!!
```

```
O:\TROJAN\VBS\SEEKER\H\VBS_000_.HTM - ARCHIVE HTML
O:\TROJAN\VBS\SEEKER\H\VBS_000_.HTM\JSCRIPT.ENCODE-0 PACKED ⤸
 BY ENCODED SCRIPT
O:\TROJAN\VBS\SEEKER\H\VBS_000_.HTM\JSCRIPT.ENCODE-0 INFECT ⤸
ED WITH TROJAN.SEEKER
```

```
k:\INTENDED\O97M\TRISTATE\M_PPT\WERGUI.PPT  archive: Embedded
k:\INTENDED\O97M\TRISTATE\M_PPT\WERGUI.PPT/macros  archive: ⤸
 Embedded PowerPoint
k:\INTENDED\O97M\TRISTATE\M_PPT\WERGUI.PPT/macros/macros    ⤸
infected: Macro.Office.Triplicate.m
```

```
K:\INTENDED\O97M\TRISTATE\M_PPT\WERGUI.PPT - archive POWERP ⤸
OINT
>K:\INTENDED\O97M\TRISTATE\M_PPT\WERGUI.PPT\Storage0 infect ⤸
ed with W97M.Sugar
```

In dieser Darstellungsweise lässt sich auch gleich das nächste Problem erkennen:

(iii) Archiv-Dateien werden selber oft als nicht infiziert bzw. als „Ok“ gemeldet, auch wenn die Archiv-Inhalte als infiziert gemeldet werden.

Dieses Phänomen lässt sich technisch meist dadurch begründen, dass diese „Ok“-Meldung sich auf den Archiv-Kopf[12] („Header“) bezieht. Trotzdem ist dies höchst verwirrend, da die Logdatei ohne weitere Erklärungen den Sachverhalt widersprüchlich darstellt, und Erläuterungen zur korrekten Interpretation in der Regel nicht vorhanden sind.

[12] Der Archiv-Kopf enthält nur allgemeine Angaben über das Archiv, wie zum Beispiel Versionsnummer und Inhaltsverzeichnis und ist somit kein Archiv-Inhalt sondern nur in derselben Datei enthalten.

(iv) Die Statusmeldungen der Produkte (also zum Beispiel die Aussage, ob eine Datei infiziert ist) sind oft verwirrend. In der Regel ist weder die Syntax noch die Semantik dokumentiert. Zusätzlich sind Virus- bzw. Malware-Namen zum Teil sehr verwirrend gewählt und erschweren das Verständnis zusätzlich. Beispiel:

```
H:\W97M\CLASS\FA\CLASS-FA.DOT  Infection: W97M/Not_a_virus ⤶
(exact)
```

(v) Mangels Dokumentation bleibt es meist dem Anwender überlassen, die Bedeutung und Relevanz der verschiedenen Einträge herauszufinden. Oft enthalten Scan-Reporte mehr als nur die offensichtlichen Statusmeldungen „infiziert“ und „nicht infiziert“, beispielsweise Angaben zu Dateitypen (etwa „archive HTML“), Laufzeitkomprimierung oder internen Dateistrukturen (zum Beispiel Namen von Makros in Office-Dokumenten). Auch Fehlermeldungen sind in Scan-Reports enthalten (und sollten dies natürlich auch), machen aber nur Sinn, wenn sie eindeutig zu verstehen oder ausreichend dokumentiert sind.

Zum Teil stellt sich sogar die Frage, ob bestimmte Angaben im Scan-Report überhaupt notwendig oder vielleicht nur für die im Anti-Virus-Produkt enthaltene Anzeigekomponente sinnvoll sind. Beispiel für ein schlecht lesbares Format:

```
210518123017,5,1,1,METEOR,Administrator,W95.Spaces.1445,N:\ ⤶
FILE\23W\MRONON\B\MSUB\5441\JAVA.JAR>>I:/FILE/23W/MRONON/B/ ⤶
MSUB/5441/EXA_012_.EXE,4,4,4,2147483904,33571876,"",1056473 ⤶
303,,0,,0,29234,0,0,0,1,1,13,20020619.005,17467,0,4,0,,{D03 ⤶
5D199-7948-4BA7-8A70-AF7A2FAF8ED7},,,,VTC,,8.0.9374
```

Der Report in diesem Beispiel basiert auf dem CSV-Format und wurde von dem Benutzer „Administrator“ auf dem Rechner „Meteor“ in der Domain „VTC“ erstellt. Somit lässt sich die Bedeutung einiger Felder immerhin erraten. Die Mehrzahl der Angaben sind jedoch nicht interpretierbar. Negativ fällt auch auf, dass der Report keine direkte Aussage enthält, ob eine Datei infiziert ist (dies lässt sich nur durch die Anwesenheit eines Virusnamens erraten).

Es werden von einigen Produkten aber auch ganz andere Datenformate genutzt. Diese sind meist jedoch auch nicht besser geeignet (oder schlecht implementiert). Einige Produkte können optional Reports im XML-Format ausgeben. Dadurch können zwar viele der eben beschriebenen Probleme vermieden werden, allerdings werden diese Möglichkeiten nicht unbedingt ausgenutzt.

Zum Beispiel kann in Avast auch eine XML-Ausgabe gewählt werden. Es werden dann pro Scan 5 XML-Dateien, 2 Grafiken und 1 XSL-Datei[13] erzeugt. Mangels Dokumentation ist daher nicht sofort ersichtlich, welche Datei für die Anzeige (zum Beispiel in

[13] „XML Stylesheet Language“ wird in diesem Fall zur Transformation der XML-Dateien in eine HTML-Ausgabe verwendet.

einem XML-fähigen Browser) gedacht ist. Da sich alle Informationen über die gescannten Dateien trotzdem in einer XML-Datei befinden, ist dieser Aufwand überflüssig (die restlichen XML-Dateien enthalten den Namen des Scans, eine Zusammenfassung der Ergebnisse und Copyright-Hinweise). Die XML-Dateien benutzen den Zeichensatz „`windows-1250`“, also die osteuropäische Variante des Microsoft Windows ANSI-Zeichensatzes. Obwohl dieser sicherlich auf vielen Systemen verarbeitet werden kann, ist dies nicht garantiert, da XML-Parser diesen Zeichensatz nicht notwendigerweise unterstützen müssen[14]. Ein deutlicher Vorteil ist die Trennung der Datei-Identifikation von der Statusmeldung des Scanners (und somit die eindeutige Darstellung von Dateinamen, die Leerzeichen enthalten).

Für die Darstellung der Scannermeldungen (als Wert des Attributs „`Status`“) sind keine Vorteile ersichtlich, da diese (gegenüber dem alternativen ASCII-Format) unverändert bleiben. Es hat den Anschein, als ob sogar der abschließende Zeilenumbruch des ASCII-Reports in den Wert des „`Status`“-Attributs übernommen wird. Das ist natürlich nicht sinnvoll, da es die Lesbarkeit verringert, wirkt sich aber auf die Verarbeitbarkeit der XML-Datei nur geringfügig aus.

Avast - ASCII-Format

```
P:\AC2.ACE [+] is OK
P:\GZ.GZ\P:\TAR.TAR\malw\MROW\MRONON\W\23W\PIZEROLP.XE\MROW
\234012\EXA_000_.EXE [L] Win32:ExploreZIP [Wrm] (0)
```

Die Darstellung des Dateinamens als Element-Inhalt hat den Vorteil, alle Unicode-Zeichen verwenden zu können. Allerdings hat diese Implementation die (nicht dokumentierte) Eigenart, jeweils ein Leerzeichen vor und nach dem eigentlichen Element-Inhalt einzufügen, die leicht als Teil des Dateinamens angesehen werden könnten. Bei längeren Dateinamen wird zudem nach ca. 80 Zeichen ein Leerzeichen eingefügt (in der letzten Zeile des Beispiels vor „`EXA_000_.EXE`“). Dies kann nur als Fehler in der Implementation gedeutet werden. Weiterhin ist ersichtlich, dass eine Trennung zwischen Dateien und Datei-Inhalten auch in diesem Format nicht möglich ist.

Avast - XML-Format

```
<Row DT="3E659309" Time="7:02:49 AM" Status=" [+] is OK
" StID="1"> P:\AC2.ACE </Row>
<Row DT="3E6593E1" Time="7:06:25 AM" Status=" [L] Win32:Exp
loreZIP [Wrm] (0)
" StID="0"> P:\GZ.GZ\P:\TAR.TAR\malw\MROW\MRONON\W\23W\PIZE
ROLP.XE\MROW\234012\ EXA_000_.EXE </Row>
```

[14] Immerhin ist eindeutig feststellbar, ob die Datei-Inhalte korrekt gelesen werden können.

3 Weitere Anwendungsmöglichkeiten

Über den Einsatz in der aVTC-Testauswertung hinaus könnte das hier entworfene Datenformat natürlich auch direkt von Anti-Malware-Produkten verwendet werden. Das Datenformat muss daher in der Lage sein, alle notwendigen Informationen zur Identifikation und Klassifikation von Malware aufnehmen zu können. Dabei wird im Folgenden unter Identifikation nur die Entscheidung verstanden, ob ein untersuchtes Datenobjekt Malware ist oder nicht. Demgegenüber dient die Klassifikation zur genaueren Angabe, um was für eine Art von Malware es sich handelt.

Aber auch verwandte Produkte (wie z. B. IDS bzw. „Intrusion Detection Systems“ oder „Vulnerability-Scanner“) geben in der Regel Informationen über „auffällige“ Software in Form von Logdateien weiter. Um für solche Anwendungen geeignet zu sein, müssen also auch ganz andere Daten berücksichtigt werden.

Sicherlich kann ein einheitliches Format aber helfen, sicherheitsrelevante Meldungen über Software eindeutig darzustellen und den Lernaufwand zum Verständnis solcher Meldungen zu verringern.

Darüber hinaus ist es auch denkbar, jegliche Art von Software zu klassifizieren. Auch dies könnte mittels eines geeigneten Datenformats wesentlich einfacher zu realisieren sein.

3.1 Malware

3.1.1 Identifikation von Malware

Malware wird heutzutage meist immer noch mittels Suchmuster-basierten Verfahren identifiziert. Dabei werden die zu untersuchenden Datenobjekte einfach nach so genannten „Scan strings“[1] durchsucht. Ein „Scan string“ besteht dabei aus einer Folge von Bytes, die für eine bestimmte Malware-Variante charakteristisch ist, also in jedem (oder zumindest fast jedem) Sample dieser Variante vorhanden ist und somit eine Identifikation dieser Variante erlaubt. Das Problem besteht jedoch darin, einen „Scan string“ zu finden, der tatsächlich in jedem Sample einer Malware-Variante vorkommt und gleichzeitig in keinem

[1] Andere gebräuchliche Begriffe sind „search string“, „pattern“ oder leider auch „signature“.

anderen Datenobjekt enthalten ist. Deshalb werden „Scan strings“ oft um „wildcards“ (also Platzhalter für beliebige Zeichen) ergänzt oder auch mit zusätzlichen Einschränkungen, wie zum Beispiel der Position des „Scan strings“ im Datenobjekt oder der Größe des Datenobjekts, versehen.

Darüber hinaus werden aber auch „heuristische“ sowie „generische“ Erkennungsmethoden eingesetzt. Generische Verfahren zielen darauf ab, mit einem „Scan string“ möglichst viele verwandte Malware-Varianten zu erkennen, eventuell sogar eine ganze Malware-Familie.

Heuristische Verfahren sollen dagegen eher zur Erkennung bisher unbekannter Malware-Varianten dienen. Im Gegensatz zu den Suchmuster-basierten Verfahren wird dabei nicht versucht, Malware-Varianten möglichst exakt zu identifizieren sondern eine potentielle Infektion (bei Viren) bzw. eine potentielle Schadwirkung (bei nicht-viraler Malware, z. B. die Aktivierung eines Keyloggers) festzustellen. Dies ist meist nur durch Analyse bzw. Emulation der Ausführung des untersuchten Datenobjekts möglich. Dieser Ansatz hat zwangsläufig eine wesentlich höhere Fehlerrate (insbesondere der „false positives“) zur Folge.

3.1.2 Klassifikation von Malware

Leider ist die Benennung (und damit verbunden auch die Klassifikation[2]) von Malware heutzutage nicht einheitlich.

Da keine generelle Methode zur systematischen Benennung von Malware existiert, ist bei der Vergabe von Malware-Namen ein zufälliger Anteil nicht auszuschließen. Da die Benennung zudem meist unter großem Zeitdruck erfolgt, sind Namen oft schlecht gewählt und unter den verschiedenen Anti-Malware Herstellern nicht abgesprochen. So passiert es regelmäßig, dass Malware mehrere unabhängige Namen erhält, wodurch eine systematische Erfassung zusätzlich erschwert wird. Obwohl Konventionen zur Benennung von Malware existieren (zum Beispiel die CARO[3]-Konvention, s. u.), werden diese von etlichen Anti-Malware Herstellern nicht oder nur teilweise verwendet.

Trotzdem gibt es weitgehende Übereinstimmung über die Kriterien, nach denen eine Klassifikation erfolgen sollte[4]:

Malware-Typ Der Malware-Typ (in der CARO-Konvention sowie von [Bontchev 1998] und [Pfleeger 1997] übereinstimmend als „malware type“ bezeichnet) gibt Auskunft über die Art der Weiterverbreitung sowie evtl. einige weitere Eigenschaften. Während in

[2] Da die Klassifikation in der Regel durch den Malware-Namen ausgedrückt wird, werden die Begriffe „Benennung“ und „Klassifikation“ in diesem Abschnitt synonym verwendet.

[3] „Computer Antivirus Research Organisation“

[4] Die Bezeichnungen beruhen im wesentlichen auf der CARO-Konvention und den im aVTC verwendeten Begriffen.

[Bontchev 1998] etliche spezifische Typen definiert werden, wird in [Brunnstein 1999] davon ausgegangen, dass sich jede Art von Malware durch eine Kombination replikativer und trojanischer Funktionalitäten beschreiben lässt (die Funktionalitäten werden dabei durch die grundlegenden Typen „virus“, „worm“ und „trojan horse“ definiert).

Generell wird aber eine Unterscheidung in selbst-replizierende Malware (Viren und Würmer) sowie nicht selbst-replizierende Malware getroffen.

Malware-Klasse („categories“ bei [Bontchev 1998]) Hiermit wird eine grobe Einteilung nach der Plattform vorgenommen, auf der die Malware ausgeführt wird. Ursprünglich wurde diese Unterteilung nur für Viren verwendet (und zum Teil auch als „Virus-Typ“ bezeichnet). Diese Angabe dient dazu, Plattformen mit ähnlichem Verhalten zusammenzufassen. Folgende Plattformen werden dabei unterschieden:

Boot Malware (im Wesentlichen nur Viren), die durch Integration in Bootsektoren während eines Bootvorgangs ausgeführt wird.

File Malware, die die Funktionalität eines Betriebssystems zum Ausführen von Dateien in einem Dateisystem ausnutzt.

Macro Diese Art von Malware nutzt die Fähigkeit bestimmter Anwendungen, sogenannte „Makros“ ausführen zu können. Ein Makro ist dabei eine in einer speziellen Makrosprache geschriebene Programmsequenz, die in die Dokumente der entsprechenden Anwendung eingebettet werden kann und von dieser Anwendung ausgeführt wird.

Macro-Malware ist aufgrund der umfangreichen Makro-Sprachen oft nicht auf eine Wirkung innerhalb eines Dokuments oder einer Anwendung beschränkt und ist dabei nicht vom Betriebssystem abhängig, da Anwendungen diese Funktionalität meist systemübergreifend zur Verfügung stellen.

Script Hierbei handelt es sich um Malware, die in einer interpretierten bzw. „Script“-Sprache vorliegt. Häufig verwendete Sprachen sind „VisualBasicScript“ (VBS) und „Javascript“ (JS).

Malware-Plattform Im Gegensatz zur Malware-Klasse wird mit dieser Angabe die verwendete Plattform genau angegeben, zum Beispiel „W97M“ für Macro-Malware, die in der in Microsoft Word 97 verwendeten Makrosprache geschrieben wurde. In [Brunnstein 1999] wird der Begriff „Plattform“ mit einer umfassenderen Bedeutung verwendet und bezieht sich auf sämtliche zugrundeliegende Hardware und Software. Hier wird damit nur die unmittelbare Grundlage einer Anwendung bezeichnet.

Des Weiteren wird Malware aufgrund von Ähnlichkeiten in der internen Struktur in „Familien“ und „Varianten“ einer Familie unterteilt.

3.1.2.1 CARO Namenskonventionen

In diesem Abschnitt wird die aktuelle Fassung der CARO-Konvention verwendet, die in [FitzGerald 2003] vorgestellt wurde. CARO-Namen entsprechen der folgenden Syntax (dabei werden die einzelnen Komponenten durch Namen in spitzen Klammern dargestellt):

```
<malwaretype>://<platform>/<family_name>.<group_name>.
<infective_length>.<subvariant><devolution><modifiers>
```

Jede der Komponenten (mit Ausnahme von `<modifier>`) darf dabei nur aus den Zeichen „`-`“, „`_`“ „`A-Z`“, „`a-z`“ und „`0-9`“ bestehen und maximal 20 Zeichen lang sein. Groß- und Kleinschreibung der Komponenten müssen gleich behandelt werden. Obwohl diese Syntax der eines URI[5] ähnelt, entspricht diese Verwendung (wie in [Gordon 2003] dargelegt wird) weder der Syntax noch der Bedeutung der in RFC 2396 definierten URI Schemata.

`<malwaretype>` bezeichnet den im vorherigen Abschnitt beschriebenen Malware-Typ. Die Liste der erlaubten Angaben für `<malwaretype>` ist fest vorgegeben, kann sich aber im Laufe der Zeit verändern und umfasst zurzeit: `virus` (beinhaltet auch Würmer, für die es keinen separaten Typ gibt), `intended`, `trojan`, `joke`, `generator`, `dropper`.

Für die Angabe der Malware-Plattform (`<platform>`) gibt es ebenfalls eine Liste, die allerdings regelmäßig erweitert werden muss. Es ist zusätzlich durchaus üblich, für Plattformen (festgelegte) Abkürzungen zu verwenden (zum Beispiel „`W97M`“ für „`Word97Macro`“).

Malware-Typ und -Plattform dürfen auch mehrfach angegeben werden, da dies zum Beispiel für „multipartite“ Malware erforderlich ist. Dies ist als in geschweiften Klammern eingefasste und komma-getrennte Liste von Werten realisiert worden (zum Beispiel „`{W97M,X97M}`“).

Die Malware-Familie wird durch `<family_name>` angegeben und stellt die Hauptkomponente des Malware-Namens dar. Da es keine systematische Methode zur Vergabe des Familiennamens gibt, werden diese zum Teil recht willkürlich vergeben. Dies stellt somit das größte Hindernis für einheitliche vergebene Malware-Namen dar.

`<group_name>` wird heutzutage eigentlich nicht mehr verwendet und diente bei vielen Viren auf der Plattform „DOS“ zur Gruppierung ähnlicher Varianten einer Virus-Familie. Diese müssen natürlich noch beschrieben werden können, weshalb diese Komponente nicht entfernt wurde. Die Verwendung ist aber optional.

`<infective_length>` ist eine der beiden Alternativen zur Angabe der Malware-Variante. Diese wird nur für bestimmte Arten von Viren verwendet und gibt die Größe der Infektionsroutine an.

[5] „Uniform Resource Identifier“ ist in [RFC 2396] definiert und beschreibt eine allgemeine Syntax zur Adressierung von Ressourcen in Netzwerken

Die andere (häufiger verwendete) Alternative `<subvariant>` wird als Buchstabenfolge ausgedrückt, beginnend bei A, B, .. bis Z und wird (wenn notwendig) mit AA, ... fortgesetzt. Bei dieser Art der Varianten-Angabe kann zusätzlich mit einer abschließenden Ziffer eine `<devolution>`[6]-Komponente der Malware angegeben werden, zum Beispiel A2. Diese wird als Teil der Malware-Variante angesehen.

`modifier` bezeichnet hingegen keine wirkliche Komponente des Malware-Namens, sondern ist eine Zusammenfassung optionaler Angaben, die zur Betonung bestimmter Eigenschaften an den Malware-Namen angehängt werden dürfen.

Folgende Angaben können in dieser Reihenfolge angehängt werden:

`:<locale_specifier>`

`<locale_specifier>` ist ein (zweistelliger) Sprach- oder Landescode (nach ISO 639 bzw. ISO 3166) und bedeutet, dass die Malware nur unter einer bestimmten Sprach- oder Lokalversion der Malware-Plattform lauffähig ist. Dies ist bei Macro-Malware aufgrund lokalisierter Makrosprachen häufig der Fall. Zusätzlich erhält der spezielle Code `:Uni` die Bedeutung, dass die entsprechende Malware nur auf der Unicode-Version einer Plattform lauffähig ist.

`#<packer>`

Dies bedeutet, dass die Malware komprimiert oder verschlüsselt ist. `<packer>` gibt dabei den Namen des Laufzeitentpackers bzw. -entschlüsslers an.

Ein angehängtes `@m` bedeutet, dass es sich um „mailing malware“, und `@mm`, dass es sich um ein „mass mailing malware“ handelt.

`!<vendorspecific\_comment>`

Mit dieser relativ neuen Angabe hat jeder Hersteller die Möglichkeit, eine weitere eigene Erweiterung zu verwenden, ohne die Syntax zu verletzen.

3.2 Klassifikation von Software

Die bisher beschriebenen Konzepte zur Klassifikation von Software beschränkten sich auf den Teilbereich der Malware. Dabei werden aber meist nur „verdächtig aussehende“ Datenobjekte daraufhin überprüft, ob sie nach dem derzeitigen lokalen Kenntnisstand maliziös sind. Diese Prüfung ist nicht umfassend (es werden in der Regel nicht alle Datenobjekte überprüft, z. B. keine Grafikdateien) und umfasst auch keine weiteren funktionalen Eigenschaften der Datenobjekte.

[6] Der Begriff „Devolution“ ist von „De-Evolution“ abgeleitet und bezeichnet Malware-Varianten, die durch eine nicht einwandfreie Replikation - also letztlich Fortpflanzungsfehler - entstanden sind.

3.2.1 Integritätsprüfung

Es gibt aber auch das Konzept der Integritätsprüfung. Dabei wird davon ausgegangen, dass sich ein System anfänglich in einem vertrauenswürdigen Zustand befindet. Jede spätere Zustandsänderung stellt somit eine potentielle Systemverletzung dar und muss geprüft werden. Um den vertrauenswürdigen Systemzustand festzuhalten, muss dabei der Zustand aller auf dem System vorhandener Software festgehalten werden (in der Regel mittels Prüfsummen, denkbar ist auch die Verwendung kryptographischer Signaturen). Ein deutlicher Vorteil im Vergleich zu den Verfahren der Malware-Identifikation ist der umfassendere Ansatz. Es werden nicht nur (potentiell) schädliche Datenobjekte betrachtet, sondern alle Datenobjekte eines Systems. Somit lassen sich auch Aussagen über den Gesamtzustand des Systems treffen, wozu reine Malware-Identifikations-Produkte nicht in der Lage sind. Andererseits können nur Zustands-Änderungen festgestellt werden. Das entspricht nicht einer Klassifikation als Malware, sondern diese muss weiterhin gesondert erfolgen.

Ein weiterer wesentlicher Vorteil ist, dass nicht nur Zustandsänderungen durch Malware (also intentionale Schädigungen) sondern auch Zustandsänderungen durch andere Ursachen (nicht-intentionale Schädigungen, z.B. durch Programmfehler) erkannt werden können.

Dies ist zugleich auch der größte Nachteil solcher Verfahren, da der Gesamtzustand eines heutigen Computersystems immer häufigeren Änderungen unterworfen ist (z. B. durch security updates, schnellere Entwicklungszyklen, ...) und es zugleich schwierig ist, vom Benutzer gewünschte Änderungen von nicht autorisierten Änderungen zu unterscheiden.

Diese Verfahren wurden einige Zeit lang als wesentlich vielversprechender als die Suchmuster-basierten Verfahren zur Malware-Identifikation angesehen, werden jedoch immer noch wesentlich seltener eingesetzt. Heutzutage werden solche Integritätsprüfungen meist als Teil eines „Intrusion Detection Systems“ eingesetzt.

Ein weiteres Problem, das den tatsächlichen Einsatz erschwert, ist die Speicherung der Integritätsdaten. Um überhaupt von Nutzen zu sein, müssen die Integritätsdaten besser geschützt sein als die von dem Integritätsprüfungssystem abgesicherten Daten. Wird ein Integritätsprüfungssystem auf einem Einzelsystem eingesetzt und die Daten zur Integritätsprüfung im System gespeichert, sind diese natürlich Teil des Gesamtsystems und somit relativ nutzlos. Es lässt sich bei einer Änderung des Systemzustands (z. B. bei einem Systemabsturz) nicht unterscheiden, ob die Systemdateien oder die Integritätsdaten verändert worden sind. Eine Lösung diese Problems besteht in der Sicherung der Integritätsdaten auf einem externen, zuverlässigen Datenträger (z. B. einer CDROM). Allerdings ist eine Aktualisierung der Integritätsdaten dann wesentlich aufwändiger und meist nicht mehr praktisch durchführbar.

Etwas anders sieht es bei einem Einsatz in einem Netzwerk aus. Fungiert ein einzelner Rechner als vertrauenswürdiger Speicher der Integritätsdaten für das gesamte Netzwerk,

ist nicht viel gewonnen, da im Falle eines Ausfalls oder Angriffs dieses Rechners die Integrität des gesamten Netzwerks nicht mehr überprüft werden kann („single point of failure“). Allerdings gibt es auch die Möglichkeit zweier sich gegenseitig überwachender Rechner, so dass auch solche Vorfälle erkannt werden können.

Eine ganz andere Lösung könnte sich aus dem in [Arbaugh 1996] beschriebenen Konzept eines „vertrauenswürdigen“ bzw. abgesicherten Bootvorgangs ergeben, das auf (recht unglückliche Weise) auch von der „Trusted Computing Group“ (TCG) vorangetrieben wird[7]. Dabei werden Prüfsummen zur Integritätsprüfung des Bootvorgangs sowie ein Hauptschlüssel, auf dem alle Prüfungen beruhen, in gesicherter zusätzlicher Hardware gespeichert. Beim Bootvorgang werden nur mittels Prüfsummen verifizierte und „vertrauenswürdige“ Codesequenzen ausgeführt und der Rechner somit in einen „vertrauenswürdigen“ Zustand überführt. Darauf könnte dann jede weitere Integritätsprüfung erfolgen. Ebenso könnte jede Manipulation der Integritätsdaten erkannt werden (mittels auf dem Hauptschlüssel basierender digitaler Signaturen).

3.2.2 Allgemeinere Klassifikationen

Theoretisch ist auch ein stärker differenziertes System zur Softwareklassifikation denkbar.

Zum einen kann Software wie bisher als „Malware“, bzw. „malicious“ eingeordnet werden. Dies umfasst unter anderem alle Viren und Würmer, aber auch jegliche Software, die aufgrund irgendeiner Definition (zum Beispiel in einer lokalen „security policy“ festgelegt) als schädlich angesehen wird.

Zum anderen kann Software aber auch als „verified“ oder „trusted“ eingeordnet werden[8]. Diese Klasse umfasst Software, die nachweisbar aus einer vertrauenswürdigen Quelle stammt und nicht modifiziert wurde. Zum Beispiel könnte es sich dabei um Systembibliotheken des Betriebssystemherstellers handeln, die mittels Signatur oder Prüfsumme nachweisbar unverändert sind und für deren Verhalten der Hersteller öffentlich garantiert. Für den hier beschriebenen Zweck reicht also eine Garantie, dass die Software nicht maliziös ist. Eine umfassende Funktionalitätsgarantie im Sinne einer formalen Verifikation ist zwar ebenfalls wünschenswert, wird hier jedoch nicht gesondert behandelt[9], da diese in der Praxis bisher kaum durchführbar ist.

Die Identifikation eines Datenobjekts als „verified“ ist dabei nur mittels Integritätsprüfung möglich. Die Verfahren der Malware-Identifikation sind dafür nicht ausreichend, da jegliche Modifikation des Datenobjekts erkannt werden muss. Nur auf diese Weise kann

[7] Die TCG ist die Nachfolgeorganisation der “Trusted Computer Platform Alliance“ (TCPA). Eine ausführliche Darstellung dieser Organisationen und der damit verbundenen Technologien ist zu umfangreich, um hier erfolgen zu können.

[8] Analog zu Malware bietet sich der Begriff „Goodware“ an. Dieser wird jedoch bereits in [Brunnstein 1999] mit einer anderen Bedeutung verwendet (als Bezeichnung für jede Art von nicht maliziöser Software).

[9] Ansonsten könnte die Klasse „verified“ in die Klassen „verified“ und „trusted“ aufgeteilt werden.

eine eindeutige Bindung eines Datenobjekts an die Informationen bzw. Garantien des Herstellers sichergestellt werden.

Diese Software-Klasse ist zurzeit zwar eher von theoretischer Natur, könnte sich jedoch als nützlich erweisen, um die zunehmende Software-Komplexität und somit auch die zunehmend schwierigere Unterscheidung zwischen erwünschter, fehlerhafter und bösartiger Funktionalität beherrschen zu können.

Ein Beispiel für diesen Klassifikations-Ansatz ist das „Known Goods“-Projekt, das für viele Unix-Distributionen eine Datenbank mit Prüfsummen (MD5 und SHA1) aller ausführbaren Dateien in ihrem Auslieferungszustand pflegt (unter `http://www.knowngoods.org` sind weitere Informationen zu diesem Projekt zu finden).

Die dritte Klasse enthält dann jegliche Software, deren Gut- oder Bösartigkeit nicht eindeutig festgestellt werden kann. Über die Software in dieser Grauzone sind keine verlässlichen Informationen vorhanden. Somit ist die Verwendung dieser „unknown“-Software aus Sicherheitsaspekten mit erheblichen Risiken verbunden. Im Prinzip lässt sich der Großteil der heutigen Software in diese Kategorie einordnen.

Somit ist die Möglichkeit zur Klassifikation jeglicher Art von Software nach möglichst umfassenden Kriterien ein wichtiges Ziel beim Entwurf des Datenformats.

4 Konzept

4.1 Anforderungen der aVTC-Tests

Syntaktische Anforderungen:

- Dateien werden im aVTC anhand ihres absoluten Pfads identifiziert. Also müssen Dateinamen und -pfade ohne Einschränkung der Länge ungekürzt verarbeitet werden können.
- Es muss jede möglicherweise vorkommende Identifikation eines Datenobjekts auch eindeutig verarbeitet werden können. Diese grundlegende Bedingung jeglicher automatisierten Verarbeitung ist zum Beispiel nicht gegeben, wenn (wie im aVTC) beliebige Dateinamen vorkommen können und nicht entsprechend gekapselt werden.

Semantische Anforderungen:

- Physikalische und logische Objekte sollen getrennt dargestellt werden. Insbesondere sollten Dateien und Datei-Inhalte nicht vermischt werden.
- Standardisierte Kategorien zur Klassifizierung der Malware sind wünschenswert. Beispielsweise sollten alle Meldungen, die eine infizierte Datei kennzeichnen, auch eindeutig als eine solche Meldung erkennbar sein.

4.2 Anforderungen für eine allgemeine Klassifikation von Software

Syntaktische Anforderungen:

- Das Datenformat muss eine eindeutige Identifikation aller enthaltenen Elemente erlauben. Dazu müssen insbesondere Dateipfade beliebiger Länge verarbeitet werden können. Dazu müssen für alle darstellbaren Datenobjekte Eigenschaften gefunden und verarbeitet werden, die eine eindeutige Identifikation erlauben. Gegebenenfalls kann dafür auch eine Kombination mehrerer Eigenschaften verwendet werden.
- Die Menge der beschreibbaren Objekte darf durch die eindeutige Identifikation nicht eingeschränkt werden.

- Keine Einschränkungen durch die Textkodierung. Wenn beispielsweise Objekte nicht mittels der ASCII-Kodierung dargestellt werden können, kann diese Kodierung nicht ausschließlich verwendet werden.

Semantische Anforderungen:

- Es sollte versucht werden, weit akzeptierte Klassifikationen bereits vorzugeben, damit diese eindeutig definiert sind. Andererseits soll die Menge der Klassifikationen erweiterbar sein, da diese nicht vollständig modellierbar ist.

4.3 Sonstige Anforderungen

Das Datenformat soll möglichst gut verständlich sein - sowohl für Menschen gut lesbar wie für Software einfach verarbeitbar. Aus diesem Grund sollte es so einfach wie möglich aufgebaut sein. Einerseits sollen möglichst viele Daten zur Identifikation und Klassifizierung berücksichtigt werden, anderseits sollen die für jedes Datenobjekt notwendigen Angaben so gering wie möglich gehalten werden.

Die wichtigsten Informationen sollten auch bei Verlust jeglicher Formatierung erhalten bleiben, um die Lesbarkeit zu erhöhen. Dies bezieht sich im Wesentlichen auf die Eigenschaft von XML, Daten sowohl als Elementinhalt wie auch als Attribut-Wert speichern zu können. Bei der Umwandlung in unformatierten Text gehen in der Regel die Attribut-Werte zusammen mit der Textformatierung verloren. Es bedeutet also, dass keine wichtigen Daten in Textauszeichnungen „versteckt“ werden sollten. Ebenso lassen sich binäre Datenformate mit dieser Forderung nicht vereinbaren.

Um einen sicheren Austausch von Klassifikationsberichten zu ermöglichen, soll es nicht möglich sein, Malware mittels dieses Datenformats zu kodieren bzw. zu transportieren. Somit darf auch keine vollständige Beschreibung aller Eigenschaften eines Datenobjekts ermöglicht werden[1].

4.4 Grundlegender Entwurf

Es wird eine Sprache entworfen, die eine Klassifikation entsprechend den in Abschnitt 3.2.2 beschriebenen Klassen „`malicious`“, „`verified`“ und „`unknown`“ ermöglicht. Da diese Unterteilung eher der Sichtweise von Anti-Malware-Produkten entspricht, werden zwei ergänzende Klassen „`modified`“ und „`update`“ eingeführt, um die für andere Anwendungen interessanten Datenobjekte deutlicher hervorheben zu können. Diese Klassen sind dabei allerdings nicht als orthogonal anzusehen, sondern als eine mit den bereits beschriebenen Klassen überlappende Erweiterung.

[1] Davon abgesehen ist es auch nicht sinnvoll, für Berichtszwecke komplette binäre Objekte zu beschreiben.

Die Klasse `modified` berücksichtigt das Konzept von Integritätsprüfern, nur Abweichungen eines bekannten Zustands zu ermitteln, ohne eine nähere Analyse auf Bös- oder Gutartigkeit eines Datenobjekts durchzuführen.

Die Klasse `update` hat einen ähnlichen Ansatz, soll aber nicht Vergleiche mit einem vorher gespeicherten Systemzustand festhalten, sondern Hinweise auf eine notwendige Zustandsänderung zu einem zukünftigen Systemzustand geben. Dies dient im Wesentlichen dazu, das „Software Update Management“ zum Beheben fehlerhafter Funktionalitäten zu unterstützen.

Die Sprache wird unter der Bezeichnung „Software Classification Language“ (SCL) entwickelt, um den allgemeineren Ansatz zu verdeutlichen.

Aus verschiedenen Gründen bietet sich ein XML-basiertes Format an (XML wird in Kapitel 5 ausführlich dargestellt):

- Aufgrund der Verwendung von Unicode können nahezu alle bekannten Zeichen bzw. Sprachen dargestellt werden.
- XML kann mit vielen verschiedenen Textkodierungen verwendet werden, da es von der konkret verwendeten Kodierung abstrahiert. Insbesondere wird keine Kodierung fest vorgegeben.
- XML verfügt über ein ausgereiftes Konzept zur Daten-Strukturierung ohne Einschränkung der Ausdrucksmächtigkeit.
- XML-Dokumente können einfach in andere Formate transformiert werden
- XML ist ein offener, frei verfügbarer Standard.
- XML-Dokumente sind portabel und leicht zu verwalten, da es sich um ein textbasiertes Format handelt (kein Binär-Format).
- XML wird bereits in vielen anderen Bereichen erfolgreich eingesetzt. Dadurch existiert bereits eine grundlegende Unterstützung zur Bearbeitung und Darstellung von XML-Dokumenten[2].

Nachteilig sind vor allem die vielfältigen Nutzungsmöglichkeiten von XML. Da es sich um eine sehr offene Sprache mit vielen Erweiterungsmöglichkeiten handelt, ist es auch möglich, Malware in ein XML-Dokument einzubetten (zum Beispiel als „processing instruction“, siehe Abschnitt 5.1), ohne dass dies in einer XML-basierten Sprache syntaktisch verhindert werden kann. Da sich allerdings die semantisch relevanten Anteile (also die Elemente und vor allem die Elementinhalte) genau festlegen lassen, wird dieser Nachteil als nicht so schwerwiegend eingeschätzt (auch wenn sich die oben genannte Anforderung somit nicht vollständig realisieren lässt). Eine direkte Einbettung von Malware als semantisch gültiger Inhalt (wie es zum Beispiel in der von Markus Schmall entworfenen

[2] Insbesondere enthalten auch viele Internet-Browser eine generische XML-Unterstützung.

Sprache „MetaMS“ [Schmall 2002] möglich ist) lässt sich somit auch in XML-basierten Sprachen ausschließen.

Im Prinzip ist auch ein direkt SGML-basiertes Format möglich[3]. Aufgrund der höheren Komplexität von SGML, die jedoch in diesem Fall nicht benötigt wird, wird dieser Ansatz nicht weiter verfolgt.

Ebenso erscheint es wenig sinnvoll, ein komplett eigenes Format auf „plain text“-Basis zu entwerfen.

[3] Da XML auf SGML beruht, ist natürlich auch jedes XML-Format SGML-basiert, wenn auch in einer vereinfachten Form.

5 Technische Grundlagen

5.1 XML

Die „Extensible Markup Language“ (abgekürzt: XML) ist ein vom W3C[1] erarbeiteter Sprachstandard und liegt als „W3C Recommendation“ zurzeit in der Version 1.0 (Second Edition) vor[2] [XML]. XML ist eine Teilmenge des SGML-Standards („Standard Generalized Markup Language“, definiert in ISO 8879:1986), die für die Verwendung im Internet optimiert wurde.

XML dient zur Definition der Syntax und der logischen Strukturen von Dokumenten. Im Gegensatz zu anderen Auszeichnungssprachen sind in XML fast keine logischen Auszeichnungselemente fest vordefiniert worden. Stattdessen werden nur relativ wenige Regeln vorgegeben, die ein XML-Dokument erfüllen muss, um als „wohlgeformt“ (engl. „well-formed“) bezeichnet werden zu können. Genau genommen definiert der XML-Standard ein „XML-Dokument“ sogar als ein Datenobjekt, das wohlgeformt ist: „A data object is an XML document if it is well-formed, as defined in this specification.“ [XML, chapter 2].

Ein XML-Dokument besteht aus Zeichendaten („character data“) und Auszeichnungszeichen („Markup“) [XML, section 2.4]. Während die Zeichendaten die eigentlichen Inhalte darstellen, dienen die Auszeichnungszeichen zur logischen und inhaltlichen Strukturierung der Inhalte. Die wichtigsten Auszeichnungen sind *Elemente*, *Attribute*, der *XML-Deklaration* sowie die Angabe einer *Dokumenttyp-Deklaration* („document type declaration“, siehe Abschnitt 5.1.1). Darüber hinaus existieren noch weitere Auszeichnungsobjekte, die hier nicht näher erläutert werden: *Verarbeitungsanweisungen* („processing instructions“), *CDATA-Abschnitte* (Abk. für „character data“, das sind nicht analysierte - engl. „unparsed“ - Textabschnitte, in denen Auszeichnungszeichen nicht interpretiert werden) und *Kommentare*.

Ein Element besteht aus einer Startmarkierung (*start-tag*), dem Element-Inhalt und einer Endmarkierung (*end-tag*) [XML, section 3.1]. Ein Start-Tag besteht aus dem Elementnamen in spitzen Klammern (zum Beispiel: `<name>`), ein End-Tag wird durch einen zusätzlichen Schrägstrich gekennzeichnet (Beispiel: `</name>`). Alternativ sind auch leere Elemente erlaubt, bei denen nur ein kombiniertes Start- und End-Tag existiert. Dies wird

[1] „World Wide Web Consortium“

[2] Die Nachfolge-Version 1.1 [XML11] existiert bereits seit einiger Zeit, hat aber erst den Status einer „W3C Candidate Recommendation“ und ist somit als noch nicht endgültig anzusehen. Diese Arbeit verwendet ausschließlich die Version 1.0 (Second Edition).

mit einem abschließenden Schrägstrich gekennzeichnet (Beispiel: `<br/>`). Einem Element können zusätzlich Attribute hinzugefügt werden. Jedes Attribut besteht aus einer Zuweisung eines Wertes an einen Namen, die im Start-Tag notiert wird. Beispiel: `<name attr1="foo" attr2="42">`.

Die XML-Deklaration steht immer ganz am Anfang eines XML-Dokuments und dient zur Angabe der verwendeten Version des XML-Standards sowie der verwendeten Zeichenkodierung [XML, section 2.8]: `<?xml version="1.0" encoding="ISO-8859-1"?>` Das Attribut „`version`“ muss immer angegeben werden (und muss zurzeit immer den Wert „`1.0`“ besitzen), während das Attribut „`encoding`“ weggelassen werden kann (es wird dann eine UTF-8 Kodierung des Unicode-Zeichensatzes angenommen). Die XML-Deklaration ist zwar optional, sollte aber nur in Ausnahmefällen weggelassen werden.

Darüber hinaus muss jedes wohlgeformte XML-Dokument folgende Regeln erfüllen[3]:

- Es muss genau ein Wurzelelement (engl. „root element“) existieren
- Die Zeichen `<` und `&` (sowie in bestimmten Situationen auch `>`) dürfen unmaskiert nur als Markup, in Kommentaren, Verarbeitungsanweisungen oder CDATA-Abschnitten vorkommen und müssen ansonsten maskiert werden, zum Beispiel durch die Zeichenfolgen `&lt;` für `<`, `&amp;` für `&` sowie `&gt;` für `>`.
- Zu jedem Start-Tag muss ein entsprechendes End-Tag mit demselben Elementnamen vorhanden sein
- Elemente dürfen sich nicht überlappen
- Kein Element darf mehrere Attribute mit demselben Namen besitzen
- Attributwerte müssen innerhalb von einfachen oder doppelten Anführungszeichen stehen. Enthält ein Wert einfache und doppelte Anführungszeichen, müssen diese maskiert werden: `'` durch `&apos;` und `"` durch `&quot;`
- Kommentare und Verarbeitungsanweisungen dürfen nicht innerhalb eines Start- oder End-Tags vorkommen (aber durchaus in Elementinhalten)

Ein wohlgeformtes Dokument ist zum Beispiel das folgende[4]:

[3] Dies ist keine vollständige Aufzählung aller Wohlgeformtheits-Regeln. Diese Liste orientiert sich an der Übersicht in [Harold 2001, S. 23], da die Regeln nicht zentral, sondern über die ganze Spezifikation verteilt definiert werden.

[4] Da im folgenden sowohl die Strukturen der XML-Dokumente wie auch die darin definierten Strukturen behandelt werden, sind die Beispiele farbig gestaltet, um die XML-Auszeichnungen hervorzuheben. Dabei werden *Elementnamen* blau, *Attributnamen* rot und *Attributwerte* grün dargestellt. Soweit die Darstellung auf diese Weise eindeutig ist, wird im Text nicht zusätzlich darauf hingewiesen.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<buch isbn="0123456789">
    <titel>Beispielbuch</titel>
    <autor>Messerschmidt &amp; andere</autor>
    <kapitel name="Einleitung">
        Dies ist der Text der Einleitung.
    </kapitel>
    <unsinn einheit="egal">
        42
    </unsinn>
</buch>
```

Wohlgeformt bedeutet also im Wesentlichen, dass die Syntax eines Datenobjekts korrekt im Sinne des XML-Standards ist, sagt aber weder etwas über die Dokumentinhalte noch über die logischen Strukturen des Dokuments aus. Deshalb gibt es auch noch den Begriff des „gültigen“ (engl. „valid“) Dokuments (der ebenfalls in [XML, section 2.8] definiert ist).

Ein gültiges XML-Dokument ist nicht nur wohlgeformt, sondern entspricht darüber hinaus auch einer vorgegebenen logischen Struktur. Auch wenn es ohne Weiteres möglich ist, nur mit wohlgeformten Dokumenten zu arbeiten, ist XML eher als Metasprache zur Definition anderer Auszeichnungssprachen zu verstehen, die dann eine dem jeweiligen Zweck angepasste logische Struktur vorgeben, ohne die zugrunde liegende Syntax und Verarbeitungsweise jedesmal aufwändig neu definieren zu müssen.

Die wohl bekannteste Anwendung von XML ist die Sprache XHTML [XHTML], eine dem SGML-basierten Standard HTML 4.01 [HTML4] vergleichbare Sprache. Obwohl die logische Struktur dieser beiden Sprachen fast identisch ist, ist XHTML aufgrund der zusätzlich geforderten Wohlgeformtheit der Dokumente wesentlich strikter definiert. Während ein HTML 4.0 Parser ein nicht gültiges Dokument in jedem Fall so gut es geht zu interpretieren versucht (mit zum Teil unvorhersehbaren Ergebnissen), bricht ein XHTML Parser ab, sobald das Dokument nicht mehr wohlgeformt ist. Dadurch gibt es auch keinen Spielraum für Aufweichungen des Standards durch proprietäre Erweiterungen bestimmter Parser, während dies bei HTML zunehmend zum Problem bei der Verarbeitung von Dokumenten geworden ist.

5.1.1 Definition logischer Strukturen

Zur Vorgabe logischer Dokumentstrukturen existieren inzwischen verschiedene Methoden.

Die älteste Methode ist die „Document Type Definition“ (DTD), die zur Verwendung mit SGML-Dokumenten entwickelt wurde. Die Syntax einer DTD ist daher nicht an XML angepasst (DTDs sind nicht wohlgeformt). Zusätzlich sind die Ausdrucksmöglichkeiten

bei Verwendung einer DTD deutlich geringer als zum Beispiel bei einem XML Schema.

„XML Schema“ (siehe Abschnitt 5.3) ist ein recht neuer Standard, der direkt für die Verwendung mit XML-Dokumenten entwickelt wurde und auch viele weitere XML-Standards berücksichtigt (zum Beispiel Namespaces, XPath und XInclude). Zusätzlich lassen sich logische Strukturen wesentlich feiner beschreiben. Zum Beispiel lässt sich in einer DTD nur festlegen, ob ein Element Subelemente, Text oder beides enthalten darf. Mit einem XML Schema könnte man auch festlegen, dass der Inhalt ein Datum oder eine Zahl sein muss; es könnte auch eine Aufzählung erlaubter Inhalte angegeben werden (etwa „ja“, „nein“). Noch werden heutzutage allerdings vorwiegend DTDs eingesetzt.

Eine direkte Konkurrenz zu XML Schema ist die Sprache RELAX NG („Regular Language description for XML - Next Generation“), eine Vereinigung der Sprachen RELAX von James Clark [Clark 2001] und TREX von Murata Makoto [Murata 2000]. Die RELAX NG Spezifikation ist bei der OASIS[5] entwickelt und veröffentlicht worden [RelaxNG]. Weitere Informationen sind auf der WWW-Seite `http://www.relaxng.org` zu finden.

RELAX NG ist eine etwas neuere Methode als XML Schema, die jedoch einige umstrittene Ansätze von XML Schema anders löst und dadurch eine höhere Ausdrucksmächtigkeit erlangt. Angeblich ist die Effizienz trotzdem mit der von XML Schema vergleichbar (insbesondere soll die Validierung mittels endlicher Automaten ebenfalls in linearer Zeit möglich sein). Allerdings wird RELAX NG noch nicht besonders gut unterstützt, während XML Schema bei neueren Entwicklungen (wie zum Beispiel „web services“) bereits eine Standard-Technologie geworden ist. Dies könnte sich jedoch durchaus ändern, da RELAX NG am 28. November 2003 von der ISO[6] standardisiert worden ist [ISO 19757-2].

Es gibt noch etliche weitere (zum Teil auch im Einsatz befindliche) Methoden, die hier nicht näher beschrieben werden können, unter anderem Schematron[7], XML-Data-Reduced (XDR)[8] und Schema for Object-Oriented XML [SOX].

Als der XML-Standard entwickelt wurde, existierten allerdings nur DTDs. Daher wurde eine gegenüber den mit SGML verwendeten DTDs vereinfachte Syntax direkt in die XML-Spezifikation aufgenommen. Dadurch definiert die XML-Spezifikation zwar Standards sowohl für die Syntax wie für die logische Struktur eines XML-Dokumentes, allerdings ist die Verwendung alternativer Methoden dadurch etwas komplizierter.

Einem XML-Dokument wird eine bestimmte DTD (und somit eine logische Struktur) zugeordnet, indem vor dem ersten Element (also direkt nach dem XML-Prolog) eine *Dokumenttyp-Deklaration* einfügt wird. Diese kann entweder die zu verwendende DTD

[5] „Organization for the Advancement of Structured Information Standards“
[6] „International Organization for Standardization“
[7] `http://www.ascc.net/xml/schematron/`
[8] `http://www.ltg.ed.ac.uk/ ht/XMLData-Reduced.htm`

direkt (intern) oder nur einen Verweis auf eine externe DTD enthalten (Kombinationen dieser beiden Möglichkeiten sind ebenfalls erlaubt). Beispiel:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
    <head>
        <title>Hello world</title>
    </head>
    <body>
        <h1>Hello world</h1>
    </body>
</html>
```

Eine Dokumenttyp-Deklaration beginnt mit den Zeichen `<!DOCTYPE` und enthält als erstes den Namen des Wurzelelements (hier: `html`). Danach folgen entweder in eckigen Klammern Deklarationen in der DTD-eigenen Syntax oder ein Verweis auf ein externe Ressource. Dies kann entweder mit dem Schlüsselwort `SYSTEM` als Datei auf dem eigenen System angegeben werden oder es kann mittels des Schlüsselworts `PUBLIC` auf eine öffentlich verfügbare Ressource verwiesen werden (eventuell mit zusätzlicher Angabe einer URL). In diesem Beispiel wurde auf die per `http` erreichbare DTD des XHTML 1.0 Standards verwiesen.

Möglich wird die Verwendung alternativer Methoden nur, weil die Angabe einer DTD in einem XML-Dokument optional ist. Aus Sicht der XML-Spezifikation kann das XML-Dokument dann aber nicht mehr gültig sein. Daher müssen in zusätzlichen Standards äquivalente Begriffe und Validierungsverfahren definiert werden. Ebenso muss eine validierende Instanz nicht nur die XML-Spezifikation, sondern auch noch alle zusätzlich erforderlichen Spezifikationen (zum Beispiel die XML Schema Spezifikation) implementieren.

Wird ein XML-Dokument gegen ein XML Schema validiert, wird dies deshalb genau genommen als „schema-gültig“ („schema valid“) bezeichnet. Im Folgenden wird dies aber nicht unterschieden, sondern mit „gültig“ nur die Übereinstimmung mit einer logischen Struktur bezeichnet, unabhängig davon mit welcher Methode diese beschrieben wurde.

Ebenso wird nicht weiter auf die Möglichkeiten eingegangen, die sich durch gleichzeitige Verwendung eines XML Schemas und einer DTD ergeben, da solche Spezialfälle die praktische Anwendung in der Regel unnötig erschweren.

5.2 XML Namespaces

Der angedachte weit verbreitete Einsatz von XML hat langfristig zur Folge, dass sich Kollisionen von Elementnamen in XML-Dokumenten nicht vermeiden lassen. Deshalb wurde zusätzlich das Konzept der „Namespaces“ (Namensräume) eingeführt [XMLNS]. Durch diese Erweiterung des XML-Standards wird die gemeinsame Verwendung mehrerer XML-Standards in einem Dokument erheblich erleichtert und somit eine bessere Modularität und Wiederverwendbarkeit von XML-Dokumenten ermöglicht.

Dazu müssen allerdings die im XML-Standard definierten Syntax-Regeln für Elementnamen leicht abgeändert werden. Ein Elementname besteht jetzt aus einem Präfix und einem lokalen Teil, die durch ein ‘:’ getrennt werden. Somit darf der Doppelpunkt im lokalen Teil nicht mehr enthalten sein.

Jedem Präfix wird ein „Uniform Resource Identifier“ (URI, definiert in [RFC 2396]) zugeordnet, der den Namensraum eindeutig kennzeichnet. Das Präfix identifiziert den Namensraum also nicht direkt, sondern dient nur als eine Abkürzung. Somit können auch mehrere Präfixe auf denselben Namensraum verweisen. Im Prinzip kann für jeden Namensraum ein beliebiges Präfix frei gewählt werden, in der Praxis ist es aber üblich, bekannte Namensräume immer mit demselben Präfix abzukürzen, da dies die Lesbarkeit deutlich verbessert. Das Präfix „xml“ (sowie alle Präfixe, die mit „xml“ beginnen) ist darüber hinaus für die XML-Spezifikationen selber reserviert und darf nicht anderweitig verwendet werden.

Es ist zusätzlich zu beachten, dass ein URI nicht unbedingt eine eindeutige Syntax besitzt, sondern eventuell auch mehrere Schreibweisen für denselben URI erlaubt sind[9]. Es ist also nicht ausgeschlossen, dass zwei verschiedene Präfixe denselben Namensraum kennzeichnen, auch wenn die Zeichenketten der entsprechenden URIs nicht identisch sind.

Ein Namensraum wird in einem XML-Element deklariert, indem ein Attribut aus dem Namensraum „xmlns“[10] eingefügt wird, dessen Name dem Präfix und dessen Wert dem URI des neuen Namensraums entspricht. Der Namensraum bzw. das Präfix ist dann innerhalb dieses Elements verwendbar. Oft wird daher ein Namensraum im Wurzelelement deklariert, um diesen im gesamten Dokument verwenden zu können. Zum Beispiel wird der XML Schema Namensraum mit dem Präfix „xsd“ und dem URI „`http://www.w3.org/2001/XMLSchema`“ etwa so deklariert und verwendet:

[9] Etwa weil Variationen in der Groß-/Kleinschreibung ignoriert werden. Zum Beispiel kennzeichnen die URIs „`http://www.w3.org/`“ und „`http://WWW.w3.org`“ denselben Namensraum.

[10] Die Namensräume „xml“ und „xmlns“ müssen nicht deklariert werden, sondern können direkt verwendet werden.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation>
      <html:p xmlns:html="http://www.w3.org/1999/xhtml">
        Die Dokumentation kann im HTML-Format erfolgen
      </html:p>
    </xsd:documentation>
  </xsd:annotation>
</xsd:schema>
```

Man kann in einem XML-Dokument auch einen Standard-Namensraum angeben, indem man im Wurzelelement einen Namensraum ohne Präfix deklariert. Alle im Dokument vorkommenden Elemente ohne Präfix werden dann diesem Standard-Namensraum zugeordnet. Das entsprechende Beispiel sieht so aus:

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <annotation>
    <documentation>
      <html:p xmlns:html="http://www.w3.org/1999/xhtml">
        Die Dokumentation kann im HTML-Format erfolgen
      </html:p>
    </documentation>
  </annotation>
</schema>
```

5.3 XML Schema

Die XML Schema Spezifikation ist recht komplex und besteht daher aus einer „nicht normativen“ Einführung [XMLSchema-0], sowie der Beschreibung der Struktur von XML Schema Dokumenten [XMLSchema-1] und der Beschreibung der vordefinierten Datentypen [XMLSchema-2]. Die folgende Beschreibung beruht daher neben der Spezifikation selber auch auf der Darstellung in [van der Vlist 2002].

Ein XML Schema ist ein wohlgeformtes XML-Dokument, in dem ähnlich wie in einer DTD mittels Element- und Attributdeklarationen die erlaubten Elemente und Attribute für die zu definierende Struktur festgelegt werden. Im Gegensatz zu einer DTD stehen für Elemente jedoch mehrere unterschiedlich komplexe Inhaltsmodelle[11] sowie viele vordefinierte Datentypen zur Verfügung. Weitere Vorteile sind die Möglichkeit zur Definition

[11] Ein Inhaltsmodell beschreibt die Struktur des Element-Inhalts für einen Elementtyp.

eigener Datentypen (auch mittels Vererbung von bestehenden Datentypen), der Gruppierung von Elementen, der einfachen Einbindung anderer Schemata sowie der erweiterten Dokumentationsmöglichkeiten.

Nachteilig sind vor allem einige Konstrukte zur Abwärtskompatibilität zu SGML-Dokumenten und die zum Teil unnötig hohe Komplexität. So existieren einige Konstrukte, die nur unter hohem Aufwand und mit vielen Einschränkungen nutzbar sind, ohne jedoch nennenswerte Vorteile zu bieten. Leider wird die Validierung dadurch deutlich komplizierter.

Ein weiterer umstrittener Punkt betrifft nichtdeterministische Konstrukte. Um möglichst einfache Implementationen zu erlauben (und weil man befürchtete, dass nichtdeterministische Konstrukte nicht effizient implementierbar sind), erlaubt XML Schema ausschließlich deterministische Inhaltsmodelle der zu entwerfenden Elemente[12]. Das bedeutet konkret, dass ein Parser immer nur ein Element nach dem anderen zu betrachten braucht und die Gültigkeit eines Elements somit nicht von den nachfolgenden Elementen abhängig sein darf. Dadurch lassen sich allerdings auch etliche Ansätze nicht realisieren, die mit anderen Schema-Sprachen (wie zum Beispiel RELAX NG) eventuell möglich sind. Ein Beispiel dafür wird am Ende dieses Kapitels angegeben.

Im Folgenden wird mit „Schema-Dokument“ das XML-Dokument bezeichnet, das das Schema selber enthält, während XML-Dokumente, deren Inhalt dem Schema entsprechend strukturiert sind, als „Schema-Instanzen“ oder „Instanz-Dokumente“ bezeichnet werden.

Alle im XML Schema Standard definierten Elemente sind in einem eigenen Namensraum (`http://www.w3.org/2001/XMLSchema`) verfügbar, so dass es kein Problem ist, dieselben Elementnamen in eigenen Sprachen zu verwenden. Dieser Namensraum wird hier wie allgemein üblich immer mit dem Präfix `xsd` abgekürzt. Analog dazu existiert auch ein Namensraum für bestimmte Elemente in den Schema-Instanzen (`http://www.w3.org/2001/XMLSchema-instance`). Dieser wird im Folgenden immer mit dem Präfix `xsi` verwendet. Um Verwechslungen zu vermeiden, wird in dieser Arbeit generell das entsprechende Präfix verwendet, auch wenn dies nicht notwendig wäre.

5.3.1 Aufbau eines Schema-Dokuments

Wie aus dem vorherigen Beispiel ersichtlich, ist `<xsd:schema>` das Wurzelelement eines Schema-Dokuments. Innerhalb des `<xsd:schema>`-Elements dürfen beliebig viele Deklarationen von *Elementen*, *Attributen*, *Element-* und *Attribut-Gruppen* sowie *Datentyp-Definitionen* vorkommen[13]. Zwischen diesen Deklarationen sowie als erstes Element innerhalb jeder Deklaration sind außerdem *Dokumentationselemente* erlaubt.

[12] DTDs unterliegen übrigens den gleichen Beschränkungen, siehe [XML, Anhang E].

[13] Es gibt noch weitere Möglichkeiten, die hier weder verwendet noch näher erläutert werden: `<xsd:include>`, `<xsd:import>`, `<xsd:redefine>` und `<xsd:notation>`

Im Gegensatz zu einer DTD gibt es aber kein ausgezeichnetes Wurzelelement für die Instanz-Dokumente, sondern jedes der direkt unterhalb des `<xsd:schema>`-Elements deklarierten Elemente kann als Wurzelelement in Instanz-Dokumenten verwendet werden. Will man dies stärker einschränken, müssen Elemente auf andere Weise deklariert werden, etwa mittels lokaler Elementdeklarationen oder als Elementgruppen.

5.3.2 Deklaration von Elementen und Attributen

Die Deklaration von Elementen für die Schema-Instanzen mit dem `xsd:element`-Element ist recht intuitiv. Dazu ist im einfachsten Fall nur die Angabe der Attribute `name` und `type` erforderlich. Beispiel:

```
<xsd:element name="alter" type="xsd:positiveInteger"/>
```

Hierdurch wird ein Element `alter` deklariert, als dessen Inhalt ausschließlich positive ganze Zahlen erlaubt sind. Im Instanz-Dokument könnte dieses Element folgendermaßen verwendet werden:

```
<alter>42</alter>
```

Dieses Instanz-Dokument ist nicht gültig und würde von einem validierenden Parser nicht akzeptiert werden:

```
<alter>unbekannt</alter>
```

Jeder vordefinierte oder im Schema global definierte Datentyp kann als `type` verwendet werden. Alternativ kann der gewünschte Datentyp aber auch „anonym“ innerhalb des `xsd:element`-Elements definiert werden:

```
<xsd:element name="alter">
  <xsd:simpleType>
    <xsd:restriction base="xsd:positiveInteger">
      <xsd:maxInclusive value="150"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

Hier wird der im vorherigen Beispiel verwendete Datentyp zusätzlich auf den maximal zulässigen Wert `150` begrenzt.

Attribute werden ähnlich wie Elemente deklariert:

```
<xsd:attribute name="sprache" type="xsd:language"/>
```

Auch hier ist eine anonyme Typdefinition möglich. Für Attribute sind allerdings nur einfache Typen als Inhaltsmodell erlaubt.

5.3.3 Definition von Datentypen

Aufgrund der vielen vordefinierten Datentypen lassen sich viele „flache“ Strukturen bereits ohne weiteren Aufwand darstellen. Allerdings sind zur Modellierung komplizierterer Strukturen bzw. geschachtelter Elemente andere Inhaltsmodelle notwendig.

In XML Schemata wird dabei zwischen *einfachen Typen* („simple types“), die weder Attribute noch Subelemente (sondern ausschließlich Text bzw. Dateninhalte) enthalten dürfen, und *komplexen Typen* („complex types“) unterschieden, für die diese Einschränkungen nicht existieren. Bei den vordefinierten Datentypen handelt es sich ausschließlich um einfache Typen. Weitere einfache Typen können durch Ableitung aus den bestehenden Datentypen definiert werden.

Komplexe Typen können wiederum unterschiedliche Inhaltsmodelle besitzen, je nachdem, ob Subelemente, Dateninhalte oder beides erlaubt sind (Attribute dürfen bei allen komplexen Typen vorhanden sein):

- „*Simple content*“ erlaubt ausschließlich Dateninhalte und ist somit nützlich, wenn man einem einfachen Datentyp Attribute hinzufügen möchte
- „*Complex content*“ erlaubt ausschließlich Subelemente
- „*Mixed content*“ erlaubt sowohl Dateninhalte wie Kindinhalte
- „*Empty content*“ erlaubt gar keine Inhalte

Neue Datentypen werden mittels der Elemente `<xsd:simpleType>` (für einfache Typen) und `<xsd:complexType>` (für komplexe Typen) definiert und können entweder global, also direkt unterhalb des `<xsd:schema>`-Elements, oder „anonym“ innerhalb einer Element- oder Attribut-Deklaration vorkommen.

5.3.3.1 Definition einfacher Typen

Einfache Typen werden durch Ableitung aus den bestehenden einfachen Datentypen definiert. Dies kann durch Einschränkung (`<xsd:restriction>`), Vereinigung (`<xsd:union>`) oder Listenbildung (`<xsd:list>`) geschehen.

Die Ableitung durch Einschränkung erfolgt dabei mittels sogenannter Facetten (engl. „facets“), die die Semantik des zugrunde liegenden Datentyps soweit möglich erhalten. Im Wesentlichen sind folgende Facetten verfügbar:

- Aufzählung erlaubter Elemente mittels `<xsd:enumeration>`
- Beschränkung der minimalen bzw. maximalen Länge von String-basierten Datentypen mittels `<xsd:minLength>` bzw. `<xsd:maxLength>`
- Angabe einer festen Länge für String-basierte Datentypen mittels `<xsd:length>`
- Angabe eines minimalen bzw. maximalen Werts für Zahlen- und Zeit-basierte Datentypen mittels `<xsd:minInclusive>`, `<xsd:minExclusive>`, `<xsd:maxInclusive>` und `<xsd:maxExclusive>`
- Beschränkung durch reguläre Ausdrücke mittels `<xsd:pattern>`

Dabei ist zu beachten, dass sich die Semantik des Basisdatentyps nicht ändern lässt. So ist es zum Beispiel durchaus möglich, mittels regulärer Ausdrücke, aus dem vordefinierten Datentyp `xsd:string` einen Typ für lokalisierte Datumsangaben zu gewinnen (es sind nämlich nur ISO 8601 Datumsformate vordefiniert). Ein selbstdefinierter Datumstyp wird intern aber immer als String behandelt und nicht wie ein Datum, ist also insbesondere nicht aufzählbar, lässt sich nicht mit anderen Datumsangaben vergleichen und es lassen sich damit auch keine Zeitperioden modellieren. Somit ist es wichtig, bei der Definition neuer Datentypen die semantischen Grenzen zu berücksichtigen und den Basis-Datentyp sorgfältig zu wählen.

Bei der Ableitung mittels Vereinigung können die zu vereinigenden Datentypen entweder als Liste im Attribut `memberTypes` angegeben oder wiederum anonym innerhalb von `<xsd:union>` definiert werden. Allerdings geht die Semantik der ursprünglichen Datentypen dabei verloren und auch deren Facetten können nicht für weitere Ableitungen von dem vereinigten Datentyp verwendet werden.

Die Ableitung mittels Listenbildung ist dahingehend eingeschränkt, dass alle Listenobjekte denselben Datentyp besitzen müssen und durch „whitespace“ voneinander getrennt werden. Der Datentyp wird mit dem Attribut `itemType` oder anonym innerhalb von `<xsd:list>` angegeben. Auch bei Listen-Datentypen geht die Semantik des Basisdatentyps verloren.

5.3.3.2 Definition komplexer Typen mit „Simple content“

Dies sind ausschließlich Ableitungen von einfachen Datentypen oder anderen komplexen Datentypen mit „simple content“. Dies kann entweder durch Erweiterung mittels `<xsd:extension>` oder durch Einschränkung mittels `<xsd:restriction>` (ähnlich der Einschränkung einfacher Typen, es kann aber zusätzlich die Verwendung von Attributen eingeschränkt werden) geschehen. Beispiel:

```
<xsd:element name="alter">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:positiveInteger">
        <xsd:attribute name="einheit" value="Jahre"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

5.3.3.3 Definition komplexer Typen mit „Complex content“

Diese Datentypen können entweder direkt definiert oder von anderen abgeleitet werden.

Eine direkte Definition beginnt mit einem der Elemente `<xsd:sequence>`, `<xsd:choice>` oder `<xsd:all>`, die wiederum beliebig viele dieser Elemente sowie Element- und Attribut-Deklarationen enthalten können.

- `<xsd:sequence>` enthält eine Folge von Subelementen mit fester Reihenfolge, die an der entsprechenden Stelle im Instanz-Dokument vorkommen müssen:

```
<xsd:element name="anschrift">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="name" type="xsd:token"/>
      <xsd:element name="straße" type="xsd:token"/>
      <xsd:element name="hausnummer" type="xsd:token"/>
      <xsd:element name="plz" type="xsd:nonNegativeInteger"/>
      <xsd:element name="ort" type="xsd:token"/>
    </xsd:sequence>
    <xsd:attribute name="land" type="xsd:token" default="D"/>
  </xsd:complexType>
</xsd:element>
```

Dieses Beispiel definiert ein Element `anschrift`, das ein Attribut `land` und fünf Subelemente enthält, die in genau dieser Reihenfolge vorkommen müssen:

```
<anschrift land="D">
  <name>ich</name>
  <straße>hier</straße>
  <hausnummer>42a</hausnummer>
  <plz>0815</plz>
  <ort>zuhause</ort>
</anschrift>
```

- `<xsd:choice>` definiert eine Auswahl. Genau eines der Subelemente muss an der entsprechenden Stelle im Instanz-Dokument vorkommen:

```
<xsd:element name="identifikation">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element name="personalausweisNr" type="xsd:token"/>
      <xsd:element name="reisepassNr" type="xsd:token"/>
      <xsd:element name="secretkeyNr" type="xsd:token"/>
      <xsd:element name="fingerprintNr" type="xsd:token"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

Und ein Instanz-Dokument:

```
<identifikation>
  <reisepassNr>0123456789</reisepassNr>
</identifikation>
```

- `<xsd:all>` enthält eine Menge von Subelementen, die in beliebiger Reihenfolge im Instanz-Dokument vorkommen dürfen. In der Praxis ist die Verwendung allerdings stark eingeschränkt, da `<xsd:all>` nicht innerhalb von `<xsd:sequence>` und `<xsd:choice>` vorkommen darf, seinerseits nur `<xsd:element>`-Elemente enthalten darf und Subelemente nicht öfter als einmal vorkommen dürfen.

`<xsd:sequence>`, `<xsd:choice>` sowie `<xsd:element>`-Deklarationen (wenn sie nicht global sind) dürfen zusätzlich mit den Attributen `minOccurs` und `maxOccurs` versehen werden, die das minimal und maximal erlaubte Auftreten des Elements hintereinander im Instanz-Dokument angeben. Es lassen sich nicht-negative ganze Werte sowie „unbounded“ als Wert für unbegrenzt häufiges Auftreten angeben. Der Standardwert ist „`1`“. Das einfache Auftreten eines Elements gilt somit als Normalfall und muss deshalb nicht explizit angegeben werden[14].

Ableitungen lassen sich ebenfalls mit den Elementen `<xsd:extension>` und `<xsd:restriction>` definieren. Leider unterscheidet sich die Semantik dieser beiden Elemente für komplexe Typen mit „complex content“ sehr von der Semantik für einfache Typen oder der für komplexe Typen mit „simple content“[15]. Dies erschwert nicht nur das Verständnis der XML Schema Spezifikation sondern ist eventuell auch gar nicht notwendig. Wie Kohsuke Kawaguchi in dem Artikel „W3C XML Schema Made Simple“ [Kawaguchi 2001] begründet, lassen sich solche Ableitungen im Wesentlichen durch Element- und Attribut-Gruppen ersetzen.

5.3.3.4 Definition komplexer Typen mit „Mixed content“

Diese Datentypen können Text und Subelement gemischt enthalten und werden als Spezialfall des „Complex content“ Modells behandelt. Durch Angabe des Attributs `mixed` im Element `<xsd:complexType>` und Zuweisung des Wertes „true“ werden gemischte Inhalte erlaubt. Die Subelemente können wie im „Complex content“ Modell definiert werden, die Textinhalte können aber weder vom Inhalt noch von der Syntax eingeschränkt werden. Vielmehr können Textinhalte im Instanz-Dokument beliebig zwischen den Subelementen eingefügt werden.

5.3.3.5 Definition komplexer Typen mit „Empty content“

Leere Elemente können recht einfach definiert werden, indem ein komplexer Typ mit „complex content“ ohne Subelemente definiert wird. Damit die Definition gültig ist, muss aber wenigstens ein Attribut deklariert sein:

[14] Aus Gründen der Übersichtlichkeit empfiehlt es sich eher, diese redundanten Angaben wegzulassen.
[15] Obwohl es sich um dieselben Elementnamen handelt, haben diese also sehr unterschiedliche Bedeutung,

```
<xsd:element name="linebreak">
  <xsd:complexType>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>
</xsd:element>
```

5.3.4 Element- und Attribut-Gruppen

`<xsd:group>` dient zur besseren Modularisierung von Typdefinitionen. Es enthält als Subelement eines der drei Elemente `<xsd:sequence>`, `<xsd:choice>` oder `<xsd:all>` und kann somit häufig verwendete Typbestandteile kapseln. `<xsd:group>` muss immer global definiert werden, kann aber von jeder Stelle, an der eines der drei anderen Elemente erlaubt ist, referenziert werden.

`<xsd:attributeGroup>` kann anstelle einer Attribut-Deklaration referenziert werden und funktioniert ansonsten genau wie `<xsd:group>`.

5.3.5 Schema-Dokumentation

Da wie bei jeder Programmiersprache eine aktuelle Dokumentation zum Verständnis der Quelltexte unabdinglich ist, stellt auch XML Schema eigene Elemente zur „Inline“-Dokumentation bereit.

Jeder Dokumentationsabschnitt besteht dabei aus einem `<xsd:annotation>`-Element, das wiederum beliebig viele `<xsd:documentation>`- und `<xsd:appinfo>`-Elemente enthalten kann. `<xsd:documentation>` ist für von Menschen gelesene Dokumentationen gedacht und kann beliebige Inhalte aufnehmen, während `<xsd:appinfo>` für die Interpretation durch andere Programme gedacht ist (und ebenfalls beliebige Inhalte aufnehmen kann).

5.3.6 Vordefinierte Datentypen

Einer der großen Vorteile des XML Schema Standards ist die Vielzahl der vorhandenen Datentypen, die in [XMLSchema-2] definiert sind[16]. Die für diese Arbeit wichtigsten Datentypen werden im Folgenden beschrieben.

[16] Dieser Teil der Spezifikation kann übrigens auch in Verbindung mit anderen Schema-Sprachen wie Relax NG verwendet werden.

string Dieser Datentyp erlaubt eine Folge beliebiger zulässiger Zeichen. Ein zulässiges Zeichen entspricht dabei der Definition in [XML, Abschnitt 2.2]: Es sind alle zulässigen Unicode-Zeichen[17] sowie die so genannten „whitespace“-Zeichen „`space`“ (ASCII-Code[18] `#x20`), „`tab`“ (ASCII-Code `#x09`), „`linefeed`“ (ASCII-Code `#x0A`) und „`carriage return`“ (ASCII-Code `#x0D`) erlaubt. Da diese Zeichen vom XML-Parser unverändert an eine Anwendung weitergegeben werden müssen, können Inhalte also auch einfache Textformatierungen mittels Tabulatoren und Zeilenumbrüchen enthalten (zum Beispiel eine Adresse).

Dies ist der einzige vordefinierte Datentyp, bei dem alle Zeichen unverändert weitergegeben werden.

normalizedString Dieser Datentyp leitet sich von `string` ab und erlaubt dieselben Zeichen als Inhalt. Allerdings werden diese vom XML-Parser nicht unverändert weitergegeben, sondern es wird eine so genannte Normalisierung („Normalization“) durchgeführt. Dabei werden alle „whitespace“-Zeichen durch „`space`“-Zeichen ersetzt. Es werden also alle Formatierungen durch Zeilenumbrüche und Tabulatoren entfernt, ohne die Länge des Inhalts zu verändern. Beispiel[19]:

```
<element>␣Dies
      ↹ist␣ein
␣␣␣␣␣␣␣␣mehrzeiliges
Beispiel␣</element>
```

wird umgewandelt zu:

```
<element>␣Dies␣␣ist␣ein␣␣␣␣␣␣␣␣␣mehrzeiliges␣Beispiel␣</element>
```

Dies ist der einzige vordefinierte Datentyp, beim dem die Länge des Inhalts erhalten bleiben muss.

token Dieser Datentyp leitet sich direkt von `normalizedString` ab, nur dass zusätzlich (wie auch bei allen anderen vordefinierten Datentypen) der Inhalt verdichtet („collapsed“) wird. Dabei werden (nach der Normalisierung) aufeinander folgende Leerzeichen jeweils durch ein Leerzeichen ersetzt. Zusätzlich werden Leerzeichen vom Anfang und Ende des Inhalts entfernt.

Da in XML-Dokumenten üblicherweise auch in Element-Inhalten Zeilenumbrüche und Leerzeichen eingefügt werden können, ohne den Inhalt zu verändern, wird für Textinhalte meist `token` verwendet.

Beispiel:

[17] Nicht-druckbare Steuerzeichen sind zum Beispiel nicht zulässig.
[18] Diese Zeichen haben identische Codenummmern in ASCII und Unicode.
[19] Tabulatoren werden als ↹, Leerzeichen als ␣ `dargestellt`

```
<element>␣Dies
      ⇥ist␣ein
␣␣␣␣␣␣␣␣mehrzeiliges
Beispiel␣</element>
```

wird umgewandelt zu:

```
<element>Dies␣ist␣ein␣mehrzeiliges␣Beispiel</element>
```

NMTOKEN ist eine Ableitung vom Datentyp `token`, steht für „name token“ und entspricht der Definition in [XML, Abschnitt 2.3]. In einem `NMTOKEN` sind nur Zeichen erlaubt, die in XML Namen vorkommen dürfen, also Buchstaben, Ziffern, ‘`.`’, ‘`-`’, ‘`_`’ und ‘`:`’. Da XML aber auf dem Unicode-Zeichensatz basiert, sind in Buchstaben und Ziffern alle Zeichen der entsprechenden Unicode-Kategorien enthalten, nicht nur die des ASCII-Zeichensatzes. Es sind somit Zeichen aus den folgenden Unicode-Kategorien zugelassen (aus [XML, Anhang B]): `Ll` („Lowercase letters“), `Lu` („Uppercase letters“), `Lo` („Other letters“), `Lt` („Titlecase letters“), `Lm` („Modifier letters“), `Mc` („Spacing combining marks“), `Me` („Enclosing marks“), `Mn` („Non-spacing marks“), `Nd` („Decimal digits“) und `Nl` („Number letters“).

Die wesentliche Einschränkung besteht also darin, dass andere Interpunktionszeichen (also Anführungszeichen, Kommata, Klammern, Fragezeichen, usw.) nicht erlaubt sind.

anyURI Dient zur Angabe einer URI gemäß RFC 2396 und RFC 2732. Allerdings darf ein Element des Typs `anyURI` auch Nicht-ASCII-Inhalte enthalten. Diese werden dann, wie im XLink-Standard definiert, vom Parser in eine RFC-konforme Darstellung umgewandelt.

positiveInteger / nonNegativeInteger `positiveInteger` dient zur Darstellung von dezimalen Ganzzahlwerten größer Null. Soll der Wert Null ebenfalls erlaubt sein, wird der Datentyp `nonNegativeInteger` verwendet.

Es ist zu beachten, dass keine obere Grenze festgelegt ist. Somit wird von einem Parser gefordert, beliebig große Werte verarbeiten zu können, obwohl dies in der Praxis natürlich nicht möglich ist.

Steht fest, dass nur ein bestimmter Wertebereich benötigt wird, sollte deshalb auch ein eingeschränkter Datentyp verwendet werden. Es existieren dabei Datentypen für die in der Informatik üblichen Wertebereich (`int`, `short`, `byte`, ...).

dateTime Definiert eine kombinierte Datums- und Zeitangabe nach ISO 8601.

Etwas vereinfacht[20] entspricht dieser Datentyp der Syntax *`YYYY`*`-`*`MM`*`-`*`DD`*`Thh:mm:ss`

[20] Die genaue Definition ist in Abschnitt 3.2.7 der Typspezifikation [XMLSchema-2] zu finden.

(wobei *YYYY* = Jahr, *MM* = Monat, *DD* = Tag, *hh* = Stunden, *mm* = Minuten, *ss* = Sekunden).

Optional können noch Bruchteile einer Sekunde (mit beliebiger Stellenanzahl) spezifiziert werden (zum Beispiel „2004-02-15T10:11:12.123456“). Ebenfalls optional kann dahinter eine Zeitzone in Form von `Z` (für UTC[21]), `+hh:mm` oder `-hh:mm` (für die Zeitdifferenz zu UTC) angehängt werden.

5.3.7 Beispiel für ein nichtdeterministisches Inhaltsmodell

Um die Einschränkungen zu verdeutlichen, die sich aus der Beschränkung auf deterministische Konstrukte ergeben, folgt hier das weiter oben erwähnte Beispiel:

```
1  <?xml version="1.0"?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3    <xsd:element name="person">
4      <xsd:complexType>
5        <xsd:sequence>
6          <xsd:choice>
7              <xsd:element name="name" type="xsd:string"/>
8              <xsd:sequence>
9                  <xsd:element name="name" type="xsd:string"/>
10                 <xsd:element name="lastname" type="xsd:string"/>
11             </xsd:sequence>
12         </xsd:choice>
13         <xsd:element name="birthday" type="xsd:date"/>
14       </xsd:sequence>
15     </xsd:complexType>
16   <xsd:element>
17 </xsd:schema>
```

Dieses Schema ist nicht gültig. Es wird darin versucht, zwei Alternativen anzugeben, mit denen sich eine Person beschreiben lässt. Zum einen kann ein Name als String (Zeile 7) und ein Geburtsdatum (Zeile 13) angegeben werden, zum anderen soll der Name auch in zwei verschiedenen Elementen als Vorname und Nachname (Zeilen 9-10) angegeben werden können. Dies funktioniert nicht, da ein Parser bei dem Element `name` nicht entscheiden kann, ob er sich in der ersten (Zeile 7) oder zweiten Auswahl (Zeile 9) befindet, ohne sich das nachfolgende Element anzusehen.

[21] „Coordinated Universal Time“, wert-identisch mit GMT

In diesem Beispiel ließe sich das vermeiden, indem man verschiedene Elementnamen verwendet (etwa `name` sowie `firstname` und `lastname`). Im Allgemeinen lässt sich jedoch nicht immer ein äquivalentes deterministisches Konstrukt finden.

6 Umsetzung des Konzepts

Für die Implementation wurde XML Schema gewählt, da die gewünschte Einschränkung der Element-Inhalte auf bestimmte Datentypen mit DTDs nicht zu realisieren ist. Da RELAX NG zu Beginn dieser Arbeit noch nicht standardisiert war und somit nicht absehbar war, ob diese Technologie überhaupt eine größere Verbreitung finden würde, wurde es ebenfalls nicht für eine Implementation in Erwägung gezogen. Eine nachträgliche Änderung schien im Vergleich zum Nutzen ebenfalls nicht sinnvoll zu sein.

Um die Verwendung mit anderen XML-basierten Sprachen zu erleichtern, wird für die „Software Classification Language“ ein eigener Namensraum mit dem URI `http://www.michel-messerschmidt.de/scl/v1` verwendet. Dabei handelt es sich um eine existierende URL, unter der auch das endgültige Schema abrufbar sein wird. Dies ist zur Identifikation des Namensraums zwar nicht notwendig, erleichtert jedoch die Validierung, da einige Validierungsprogramme zum Auffinden des Schemas auch den URI des Namensraums verwenden.

Das Wurzelelement des SCL-Schema-Dokuments sieht somit wie folgt aus:

Schema

```
<xsd:schema version="1.0"
  targetNamespace="http://www.michel-messerschmidt.de/scl/v1"
  xmlns="http://www.michel-messerschmidt.de/scl/v1"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified" xml:lang="en">
```

- Das Attribut `version` legt fest, dass die Version 1.0 der XML Schema Spezifikation verwendet wird.
- `targetNamespace` legt fest, welcher Namensraum durch dieses Schema beschrieben wird, bzw. mit welchem Namensraum die im Schema deklarierten Elemente in einem Instanz-Dokument verwendet werden dürfen.
- `xmlns` definiert den Standard-Namensraum für Elemente im Schema-Dokument, während `xmlns:xsd` den üblichen Präfix für Elemente aus dem XML-Schema-Namensraum definiert. Durch diese Kombination lassen sich alle selbst definierten

Objekte (ohne Präfix) leicht von den vordefinierten Objekten (mit Präfix `xsd`) unterscheiden.

- `elementFormDefault` wird auf den Wert „`qualified`“ gesetzt, da der Ziel-Namensraum mit dem Standardwert nicht für Elemente mit lokalen Typdefinitionen verwendet wird (diesen wird ansonsten gar kein Namensraum zugeordnet).

- Ebenso wird der Standardwert für `attributeFormDefault` explizit aufgeführt. Da Attribute laut Namensraum-Spezifikation nicht dem Standard-Namensraum angehören [XMLNS, section 5.2], sind andere Angaben nur für Spezialfälle sinnvoll.

- `xml:lang` legt die im Schema-Dokument verwendete Sprache fest.

Das Wurzelelement der Instanz-Dokumente ist das Element `report`, welches im Schema wie folgt definiert ist:

Schema

```
<xsd:element name="report">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="origin" type="originType"/>
      <xsd:element name="object" type="objectType"
        minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="version" type="xsd:token"
      default="1.0"/>
  </xsd:complexType>
</xsd:element>
```

Ein Instanz-Dokument der „Software Classification Language“ besteht syntaktisch also aus einem `report`-Element, das genau ein `origin`-Element und mindestens ein `object`-Element enthält. Semantisch muss jeder Klassifikations-Bericht also eine Quellenangabe und zumindest ein zu klassifizierendes Datenobjekt enthalten.

Zusätzlich ist für das Wurzelelement noch ein Attribut `version` definiert, das die verwendete Version des SCL-Schemas anzeigt. Ein Instanz-Dokument könnte somit wie folgt aussehen (wobei die noch zu beschreibenden Strukturen nur durch „`...`“ angedeutet werden):

Instanz

```
<report version="1.0"
  xmlns="http://www.michel-messerschmidt.de/scl/v1">

  <origin>...</origin>

  <object>...</object>

  <object>...</object>
</report>
```

Die Implementation nutzt die in XML Schema mögliche Modularisierung, um jede Ebene der Element-Hierarchie möglichst abstrakt darstellen zu können (wie zum Beispiel eben das `report`-Element ohne Kenntnis des `object`-Elements dargestellt wurde). Dies kommt gleichzeitig der Orientierung im SCL Schemadokument zugute. Die Modularisierung wird dabei weitgehend mittels globaler Datentyp-Definitionen realisiert. Alternativ hätten auch die Elemente selber global definiert werden können. Da jedoch jedes global definierte Element auch als Wurzelelement der Instanz-Dokumente erlaubt ist und darüber hinaus keine weiteren Vorteile bietet, wird diese Möglichkeit nicht verwendet[1].

Im Folgenden werden die einzelnen Elemente der „Software Classification Language“ nach ihrem logischen Zusammenhang unterteilt und nach Hierarchie-Ebenen geordnet beschrieben. Dazu werden wie eben jeweils die relevanten Abschnitte des Schema-Dokuments und gegebenenfalls Beispiele eines Instanz-Dokuments angegeben.

Das komplette Schema ist in Anhang A zu finden, ein Instanz-Dokument, das viele Anwendungsmöglichkeiten aufzeigt, in Anhang B.

6.1 Benötigte Datentypen

Außer den in XML Schema vordefinierten Datentypen werden im Folgenden auch die in diesem Abschnitt definierten eigenen Datentypen verwendet.

6.1.1 nonEmptyString

Schema

```
<xsd:simpleType name="nonEmptyString">
  <xsd:restriction base="xsd:string">
    <xsd:minLength value="1"/>
```

[1] Es gibt auch noch weitere Modularisierungsmöglichkeiten, die jedoch nur für komplexere Situationen sinnvoll sind.

```
  </xsd:restriction>
</xsd:simpleType>
```

Dies ist eine Einschränkung des vordefinierten Datentyps `xsd:string`, die die Verwendung eines leeren Inhalts verbietet, ansonsten aber die gleichen Inhalte erlaubt. Auf diese Weise kann ein leeres Inhaltsmodell für Elemente verhindert werden.

6.1.2 noSpaceToken

Schema

```
<xsd:simpleType name="noSpaceToken">
  <xsd:restriction base="xsd:token">
    <xsd:pattern value="\S*"/>
  </xsd:restriction>
</xsd:simpleType>
```

Dies ist eine weitere Einschränkung des vordefinierten Datentyps `xsd:token`, die zusätzlich noch jede Form von Leerzeichen („`space`“, „`tab`“, „`line feed`“ und „`carriage return`“) im Elementinhalt verbietet.

In dem regulären Ausdruck wird dies durch Verwendung der Zeichenklasse `\S` realisiert, die gerade alle Zeichen außer den oben genannten beinhaltet und somit die Negation der „whitespace“-Klasse `\s` darstellt[2].

6.1.3 fileURI

Dies ist eine Einschränkung des vordefinierten Typs `xsd:anyURI`, der nur das „file“-Schema erlaubt. Dies dient zur Adressierung von Dateien über ein Dateisystem und ist in RFC 1738 definiert.

Schema

```
<xsd:simpleType name="fileURI">
  <xsd:restriction base="xsd:anyURI">
    <xsd:pattern value="file://.*"/>
  </xsd:restriction>
</xsd:simpleType>
```

[2] Da viele verschiedene Syntaxregeln und Zeichenklassen für reguläre Ausdrücke existieren, implementiert jedes Programm eine unterschiedliche Teilmenge davon. XML Schema basiert auf den in perl 5 verwendeten regulären Ausdrücken.

6.1.4 hexNumber

In der XML Schema Spezifikation werden zwar numerische Datentypen definiert, allerdings nur im Dezimalsystem. Deshalb wird hier ein eigener Datentyp für hexadezimale[3] Zahlenangaben definiert.

Schema
```
<xsd:simpleType name="hexNumber">
  <xsd:restriction base="xsd:token">
    <xsd:pattern value="[0-9a-fA-F]+"/>
  </xsd:restriction>
</xsd:simpleType>
```

Es ist zu beachten, dass sowohl Klein- wie Großbuchstaben erlaubt sind. Auch beliebig viele führende Nullen sind zugelassen.

6.1.5 macAddress

Dieser Datentyp dient zur Beschreibung einer MAC-Adresse („Medium Access Control“), die zur (weltweit) eindeutigen Identifikation von Netzwerkkarten verwendet wird. Eine MAC-Adresse besteht aus 48 Bit, von denen 24 Bit einen Hersteller bezeichnen und die anderen 24 Bit als eindeutige Kennung für Geräte dieses Herstellers dienen.

Schema
```
<xsd:simpleType name="macAddress">
  <xsd:restriction base="xsd:token">
    <xsd:pattern value="[0-9a-fA-F]{12}"/>
    <xsd:pattern value="[0-9a-fA-F]{2}([-:][0-9a-fA-F]{2}){5}"/>
  </xsd:restriction>
</xsd:simpleType>
```

Um sowohl gut verarbeitbare wie leicht lesbare Angaben zu erlauben, ist einerseits die Angabe eines einzelnen hexadezimalen Werts (zum Beispiel „123456789ABC“) möglich. Anderseits darf der Inhalt auch in Form einzelner Bytes (ebenfalls hexadezimal) dargestellt werden, die durch Doppelpunkt oder Bindestrich getrennt sind (zum Beispiel „12:34:56:78:9A:BC“ oder „12-34-56-78-9A-BC“).

[3] „hexadezimal“ ist zwar eine sprachlich fragwürdige Vermischung griechischer und römischer Begriffe, wird hier jedoch trotzdem verwendet, da es in der Informatik allgemein gebräuchlich ist. Korrekterweise müsste „hexadekadisch“ verwendet werden.

6.1.6 eui64Address

Bei EUI-64 („Extended Unique Identifier“) handelt es sich um eine vom IEEE standardisierte, 64 Bit lange Kennung. Diese ist in eine 24 Bit lange Herstellerkennung und einen 40 Bit langen „extension identifier“ aufgeteilt. Des Weiteren ist in [IEEE2003] eine Methode zur Generierung von EUI-64 Adressen aus MAC-Adressen definiert.

Schema

```
<xsd:simpleType name="eui64Address">
  <xsd:restriction base="xsd:token">
    <xsd:pattern value="[0-9a-fA-F]{16}"/>
    <xsd:pattern value="[0-9a-fA-F]{2}([-:][0-9a-fA-F]{2}){7}"/>
  </xsd:restriction>
</xsd:simpleType>
```

Der Inhalt darf auf dieselbe Weise wie beim Datentyp `macAddress` angegeben werden (die Anzahl der Bytes ist natürlich unterschiedlich).

6.1.7 ipAddress

Mit diesem Datentyp wird die Syntax von IP-Adressen beschrieben (ausschließlich IP version 4).

Schema

```
<xsd:simpleType name="ipAddress">
  <xsd:restriction base="xsd:token">
    <xsd:pattern value="[0-9a-fA-F]{8}"/>
    <xsd:pattern value="(0?\d?\d|[1]\d\d|2[0-5][0-5])(\.(0?
\d?\d|[1]\d\d|2[0-5][0-5])){3}"/>
  </xsd:restriction>
</xsd:simpleType>
```

Neben der üblichen Angabe von vier dezimalen, durch Punkte getrennten Byte-Werten (zum Beispiel „192.168.1.1“), ist wiederum auch ein einziger hexadezimaler Wert erlaubt (zum Beispiel „c0a80101“ für dieselbe Adresse). Der hier verwendete reguläre Ausdruck für die dezimale Angabe ist zwar deutlich komplexer als die in Anhang B von RFC 2373 angegebene Syntax. Dafür wird es hiermit aber erreicht, ausschließlich gültige Werte zu erlauben (während die Syntax aus RFC 2373 auch einige ungültige Angaben zulässt).

6.1.8 ipv6Address

Im Gegensatz zu der recht einfachen Syntax von IPv4-Adressen, definiert RFC 2373 etliche verschiedene (hexadezimale) Schreibweisen für die 128 Bit langen Adressen von IP version 6.

Neben der Angabe von acht durch Doppelpunkt getrennten Feldern mit je 2 Byte, deren führende Nullen weggelassen werden dürfen (zum Beispiel „1080:0:0:0:8:800:200C:417A“), besteht auch die Möglichkeit eine Folge von Feldern mit Nullen durch „`::`“ abzukürzen (zum Beispiel „1080::8:800:200C:417A“). Zudem ist auch eine „abwärtskompatible“ Darstellung möglich, bei der die letzten 2 Felder (bzw. 32 Bit) durch eine IPv4 Adresse angegeben werden dürfen (zum Beispiel „1080:0:0:0:8:800:32.12.65.122“). Schließlich können diese Möglichkeiten auch kombiniert werden, zum Beispiel als „1080::8:800:32.12.65.122“ oder „1080::32.12.65.122“.

Für eine möglichst einfache Verarbeitung der Werte ist auch hier wieder zusätzlich die Angabe eines einzelnen Hexadezimal-Wertes möglich, der diesmal aus 32 Hexadezimal-Ziffern besteht.

Die Beschränkung der IPv4-Anteile erfolgt mit dem selben regulären Ausdruck wie beim Datentyp `ipAddress`:

```
(0?\d?\d|[1]\d\d|2[0-5][0-5])(\.(0?\d?\d|[1]\d\d|2[0-5][0-5])){3}
```

und wird im Schema-Ausschnitt deshalb durch das Zeichen □ abgekürzt[4].

Schema

```
<xsd:simpleType name="ipv6Address">
  <xsd:restriction base="xsd:token">
    <xsd:pattern value="[0-9a-fA-F]{32}"/>
    <xsd:pattern
      value="[0-9a-fA-F]{1,4}(:[0-9a-fA-F]{1,4}){7}"/>
    <xsd:pattern
      value="([0-9a-fA-F]{1,4}(:[0-9a-fA-F]{1,4}){0,6})?::↩
([0-9a-fA-F]{1,4}(:[0-9a-fA-F]{1,4}){0,6})?"/>
    <xsd:pattern
      value="[0-9a-fA-F]{1,4}(:[0-9a-fA-F]{1,4}){5}:□">
    <xsd:pattern
      value="([0-9a-fA-F]{1,4}(:[0-9a-fA-F]{1,4}){0,4})?::↩
([0-9a-fA-F]{1,4}(:[0-9a-fA-F]{1,4}){0,4}):□">
    <xsd:pattern
      value="([0-9a-fA-F]{1,4}(:[0-9a-fA-F]{1,4}){0,4})?::□">
  </xsd:restriction>
</xsd:simpleType>
```

[4] Während darstellungsbedingte Zeilenumbrüche wiederum durch das Zeichen ↩ gekennzeichnet werden.

Ein einzelnes 2-Byte Feld wird dabei durch den Ausdruck `[0-9a-fA-F]{1,4}` beschrieben.

6.1.9 checksumType

Auch wenn die Entwicklung neuer Prüfsummen-Algorithmen durch diesen Datentyp nicht erfasst werden kann, ist die Verwendung konstanter Werte trotzdem sinnvoll, um eine automatische Verwendung bzw. Überprüfung zu ermöglichen.

Der jeweilige Algorithmus muss dabei durch das Attribut `type` angegeben werden, während der Elementinhalt nur aus der Prüfsumme selber besteht. Für die Prüfsumme wird zwar der eigene Typ `hexNumber` für hexadezimale Werte verwendet, dies schließt aber die Verwendung dezimaler Werte mit ein.

Die hier verwendete Liste orientiert sich an der Liste der in [Schneier 1996] vorgestellten Algorithmen sowie an den Möglichkeiten der Integritätsprüfer „tripwire“ und „AIDE“. Einige dieser Algorithmen besitzen zwar bekannte Schwächen und sollten nicht mehr verwendet werden, müssen aber trotzdem beschrieben werden können.

Schema

```
<xsd:complexType name="checksumType">
  <xsd:simpleContent>
    <xsd:extension base="hexNumber">
      <xsd:attribute name="type" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:token">
            <xsd:enumeration value="SHA1"/>
            <xsd:enumeration value="MD5"/>
            <xsd:enumeration value="RIPEMD"/>
            <xsd:enumeration value="HAVAL"/>
            <xsd:enumeration value="SNEFRU"/>
            <xsd:enumeration value="GOST"/>
            <xsd:enumeration value="MD4"/>
            <xsd:enumeration value="MD2"/>
            <xsd:enumeration value="CRC32"/>
            <xsd:enumeration value="CRC16"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

6.1.10 mimeDatatype

Dieser Datentyp beschreibt die Syntax gültiger MIME-Typ-Angaben, die in Abschnitt 5.1 von RFC 2045 definiert ist. RFC 2045 definiert dazu eine feste Syntax (unter anderem die Syntax der Angabe als `Medientyp/Subtyp`) sowie die grundlegenden Medientypen, die dann in RFC 2046 näher beschrieben werden. Zusätzlich definiert RFC 2048 Prozeduren zur Registration neuer Medientypen und Subtypen.

Da an die Einführung neuer Medientypen wesentlich höhere Anforderungen gestellt werden als an die Registrierung neuer Subtypen und die Liste der Medientypen bisher nicht erweitert wurde, wird diese Liste für den hier definierten Datentyp als unveränderlich betrachtet. Sollten also in Zukunft neue Medientypen eingeführt werden, so können diese mit dem in dieser Arbeit entwickelten Schema nicht beschrieben werden[5]. Meiner Meinung nach wiegt die nur dadurch mögliche Einschränkung auf gültige MIME-Typen diesen Nachteil aber vollkommen auf. Im Gegensatz dazu wird die Liste der Subtypen nicht auf bereits vorhandene Subtypen beschränkt, da es hier recht häufig neue Entwicklungen gibt und etliche alte Subtypen existieren, die nicht den Anforderungen von RFC 2048 entsprechen.

RFC 2045 erlaubt es zusätzlich, an die `Medientyp/Subtyp`-Angabe beliebig viele (Sub-)Typ-spezifische Parameter anzuhängen. Am häufigsten verwendet werden wohl die Angabe eines Zeichensatzes bei „Text“-Typen (zum Beispiel „`text/plain; charset=iso-8859-1`“) und die Angabe der Trennzeile bei „multipart“-Typen (zum Beispiel „`multipart/mixed; boundary=gc0p4Jq0M2Yt08j34c0`“). Da die Syntax gültiger Typ-Angaben durch diese Erweiterung wesentlich komplexer wird, Parameter für die meisten Daten-Objekte hier aber nicht benötigt werden, sind Parameter in diesem Datentyp nicht erlaubt. Für Mail-Objekte existiert stattdessen eine andere Möglichkeit zur Angabe von MIME-Parametern.

Da gültige Angaben auf einer Teilmenge des ASCII-Zeichensatzes beruhen (ohne Steuerzeichen, Leerzeichen und etlichen Interpunktionszeichen), in XML-Dokumenten aber generell alle Unicode-Zeichen zugelassen sind, wird die Liste der erlaubten Zeichen hier explizit angegeben. Des Weiteren erlaubt die Definition beliebige Groß- bzw. Kleinschreibung aller Angaben.

Diese beiden Anforderungen lassen sich zwar mittels regulärer Ausdrücke erfüllen, das Resultat ist jedoch recht unübersichtlich und wird hier deshalb in vereinfachter Form dargestellt.

Ein Typ-String („token“) besteht aus mindestens einem der erlaubten Zeichen, nämlich allen Ziffern, Groß- und Kleinbuchstaben sowie etlichen Interpunktionszeichen. Dies lässt sich im XML Schema durch folgenden regulären Ausdruck beschreiben[6]:

[5] Gegebenenfalls ist also die Entwicklung und Veröffentlichung einer neuen Version dieses Schemas erforderlich.

[6] Da es sich um ein XML-Dokument handelt, muss das Zeichen & durch `&` maskiert werden.

```
[-0-9A-Za-z!#%’_‘~\$\*\+\.\^\{\|\}&amp;]+
```

Dieser Ausdruck wird im Schema-Ausschnitt durch das Zeichen □ abgekürzt.

Die beliebige Verwendung von Groß- und Kleinbuchstaben kann in regulären Ausdrücken nachgebildet werden, indem jedes Zeichen durch eine Zeichenklasse ersetzt wird. Aus dem regulären Ausdruck `text` wird dann also `[tT][eE][xX][tT]`. Diese Ersetzung wird im folgenden Ausschnitt aus Gründen der Übersichtlichkeit weggelassen, ist im Schema aber selbstverständlich vorhanden.

Schema

```
<xsd:simpleType name="mimeDatatype">
  <xsd:restriction base="xsd:token">
    <xsd:pattern value="text/□"/>
    <xsd:pattern value="image/□"/>
    <xsd:pattern value="audio/□"/>
    <xsd:pattern value="video/□"/>
    <xsd:pattern value="application/□"/>
    <xsd:pattern value="message/□"/>
    <xsd:pattern value="multipart/□"/>
    <xsd:pattern value="x-□/□"/>
  </xsd:restriction>
</xsd:simpleType>
```

6.2 Quellenangabe

Eine Quellenangabe sollte zumindest den Namen des Programms enthalten, welches das Dokument erzeugt hat. Auch eine Datums- und Zeitangabe sollte immer vorhanden sein.

Schema

```
<xsd:complexType name="originType">
  <xsd:all>
    <xsd:element name="date" type="xsd:dateTime"/>
    <xsd:element name="program" type="xsd:token"/>
    <xsd:element name="system" type="xsd:token"
      minOccurs="0"/>
    <xsd:element name="version" type="xsd:token"
      minOccurs="0"/>
    <xsd:element name="dataversion" type="xsd:token"
      minOccurs="0"/>
    <xsd:element name="options" type="xsd:string"
      minOccurs="0"/>
    <xsd:element name="contact" type="xsd:string"
      minOccurs="0"/>
  </xsd:all>
</xsd:complexType>
```

Weitere wünschenswerte Angaben sind:

- Eine Identifikation des Systems, auf dem sich die klassifizierte Software befindet, zum Beispiel durch Angabe eines Hostnamens oder einer IP-Adresse. Da solche Angaben nicht unter allen Betriebssystemen verfügbar sind, muss dies aber als optionales Element implementiert werden.
- Version und Optionen des erzeugenden Programms. Diese Angaben dienen zur besseren Einschätzung der Klassifizierung. Zum Beispiel erhöht die Verwendung von Optionen zur heuristischen Erkennung bei Anti-Virus-Programmen die Wahrscheinlichkeit von „false positive“-Erkennungen zum Teil erheblich. Da die Erfahrung im aVTC zeigt, dass viele existierende Produkte dies in ihren Logformaten ignorieren, sind diese Angaben ebenfalls optional, um eine einfache Konvertierung zu ermöglichen.

  Zusätzlich ist es höchst sinnvoll anzugeben, welche Datenbasis in welcher Version für die Klassifikation verwendet wurde (zum Beispiel das Datum der „malware pattern“ bei Anti-Malware-Software). Dieses Element (`dataversion`) ist nur optional, um die Konvertierung aus anderen Formaten zu erleichtern.
- Eine Kontaktmöglichkeit zum Hersteller des Programms kann ebenfalls sinnvoll sein, wenn zum Beispiel Unklarheiten hinsichtlich der Interpretation oder der Relevanz einzelner Klassifikationen bestehen.

Da alle Subelemente höchstens einmal vorkommen, ist es möglich, hier eine `xsd:all`-Gruppierung zu verwenden. Somit können die Subelemente von `origin` in beliebiger Reihenfolge vorkommen. Bei allen anderen Elementen (genauer gesagt: bei allen Datentypen, deren Struktur auf `xsd:sequence` oder `xsd:choice` beruht) ist die Reihenfolge dagegen fest vorgeschrieben.

Da die Inhalte der oben genannten Elemente nicht näher spezifiziert sind, ist es nicht sinnvoll, diese durch spezielle Datentypen künstlich zu beschränken. Somit wird für die meisten Angaben `xsd:token` verwendet.

Da es für Kontaktadressen und eventuell auch für Optionen nicht wünschenswert ist, Zeilenumbrüche zu entfernen, wird hierfür der Datentyp `xsd:string` verwendet.

Für die Datums- und Zeitangabe ist es hingegen sinnvoll, einen speziellen Datentyp anzugeben, um eine automatisierte Verarbeitung zu ermöglichen. Hierfür bietet sich der vordefinierte Datentyp `xsd:dateTime` an (siehe auch Abschnitt 5.3.6), der syntaktisch und semantisch dem Standard ISO 8601 entspricht[7].

Da die Angabe einer Zeitzone bei diesem Datentyp weggelassen werden kann, gibt es jedoch einige Schwierigkeiten beim Vergleichen von Zeitangaben zu berücksichtigen. Zum Einen geht die XML Schema Spezifikation (in Abschnitt 3.2.7.3 von [XMLSchema-2]) implizit davon aus, dass Zeitangaben ohne Zeitzone in Vergleichen dieselbe Zeitzone bezeichnen. Zum Anderen ist ein Vergleich zwischen Zeitangaben mit und ohne Zeitzone nicht immer eindeutig entscheidbar. Dazu wird in der Spezifikation gefordert, dass der Vergleich nur dann erfolgreich ist, wenn er auch unter Berücksichtigung eines Spielraums von 14 Stunden eindeutig entscheidbar ist.

Aufgrund dieser Schwierigkeiten wurde in den ersten Entwürfen ein eigener Datentyp für Zeitangaben entworfen, der die Angabe der Zeitzone erforderte. Dieser Datentyp wurde jedoch wieder verworfen, da Betriebssysteme existieren (unter anderem das für Notfall-Bootmedien immer noch wichtige DOS), bei denen die lokale Zeitzone nicht immer ermittelt werden kann. Stattdessen wurde mit Hilfe der Dokumentation festgelegt, dass die Angabe einer Zeitzone nur weggelassen werden sollte, wenn diese unbekannt ist.

Die Möglichkeit zur Verwendung lokalisierter Datumsangaben wurde aufgrund der in Abschnitt 5.3.3.1 erwähnten Probleme nicht weiter verfolgt.

Instanz

```
<origin>
  <date>2004-01-15T23:09:00+01:00</date>
  <system>192.168.42.8</system>
  <program>ClassifyMe</program>
  <version>2004 1.01</version>
  <dataversion>Dat1: 2004-01-14, Dat2: 2004-01-13</dataversion>
  <options>-v -N</options>
  <contact>www@michel-messerschmidt.de</contact>
</origin>
```

[7] Dasselbe Format wird in den HTML- und XHTML-Standards verwendet.

6.3 Datenobjekte

Für Datenobjekte sind zwei verschiedene Inhaltsmodelle notwendig. Entweder besteht ein Datenobjekt aus einer Identifikation (Subelement `identification`) und einer Klassifikation (Subelement `classification`) oder aus einer Identifikation und einem Inhalt (Subelement `content`). Diese Auswahl lässt sich einfach durch eine Schachtelung eines `xsd:choice`-Elements in einem `xsd:sequence`-Element verwirklichen.

Schema

```
<xsd:complexType name="objectType">
  <xsd:sequence>
    <xsd:element name="identification" type="identificationType"/>
    <xsd:choice>
      <xsd:element name="content" type="contentType"/>
      <xsd:element name="classification"
        type="classificationType"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

Da Inhalte wiederum aus Datenobjekten bestehen (siehe Abschnitt 6.3.3), ist es auf diese Weise einfach, beliebige Verschachtelungen unterschiedlicher Datenobjekte zu modellieren.

Es ist zu beachten, dass es nicht möglich ist, Inhalte und eine Klassifikation anzugeben. Diese Entscheidung wurde getroffen, um eine Konzentration auf die relevanten Datenobjekte zu betonen. Andernfalls wäre es zu leicht möglich, einen Bericht mit vielen unnötigen Details zu füllen. Auch wenn es in einigen Fällen sicherlich sinnvoll ist, eine Gesamtklassifikation anzugeben (etwa für eine ZIP-Datei), kann diese durchaus aus den Einzelklassifikationen der Inhalte abgeleitet werden und muss nicht unbedingt zusätzlich im Bericht notiert werden. Zusätzlich ist zu bedenken, dass die Bildung einer Gesamtklassifikation situationsabhängig sein kann.

Die Struktur der Klassifikationsdaten ist für alle Datenobjekte gleich, während die Identifikationdaten je nach Typ des Datenobjekts unterschiedlich strukturiert sind. Dabei wird eine Auswahl möglicher Datenobjekts-Typen fest vorgegeben. Durch die Existenz eines generischen Typs lassen sich jedoch trotzdem alle denkbaren Datenobjekte darstellen.

Instanz

```
<object>
  <identification>...</identification>
  <content>...</content>
</object>

<object>
  <identification>...</identification>
  <classification>...</classification>
</object>
```

6.3.1 Identifikation

Je nach Art des Datenobjekts sind natürlich recht unterschiedliche Identifikationsangaben notwendig. Deshalb wird für jede Art von Datenobjekt ein Subelement von `identification` mit einer entsprechend angepassten Struktur definiert.

Schema

```
<xsd:complexType name="identificationType">
  <xsd:choice>
    <xsd:element name="file" type="fileIdentType"/>
    <xsd:element name="mail" type="mailIdentType"/>
    <xsd:element name="sector" type="sectorIdentType"/>
    <xsd:element name="packet" type="packetIdentType"/>
    <xsd:element name="memory" type="memoryIdentType"/>
    <xsd:element name="octetstream" type="streamIdentType"/>
  </xsd:choice>
</xsd:complexType>
```

Folgende Datenobjekte werden unterschieden:

file Dateien

mail Email bzw. Nachrichten nach dem MIME-Standard. Eine Email kann zwar auch als Datei vorliegen und identifiziert werden, da oft jedoch nur bestimmte Teile (zum Beispiel ein viraler Anhang) von Interesse sind, ist eine daran angepasste Identifikation sinnvoll.

sector Eine Gruppe von Sektoren einer Festplatte, CDROM, Diskette oder eines anderen Datenträgers. Dies ist für Datenobjekte notwendig, die nicht über ein Dateisystem adressierbar sind. Ein typisches Beispiel sind Bootviren.

Natürlich ist auch jede Datei als Gruppe von Sektoren beschreibbar. Da dies jedoch nur die Menge der Identifikationsdaten erhöht, ohne irgendeinen Vorteil zu erbringen, sollte darauf verzichtet werden.

packet Ein oder mehrere Netzwerkpakete. Auf diese Weise lässt sich jede Art von Netzwerkverkehr beschreiben, auch wenn dieser nicht auf einem Datenträger gespeichert wird.

Für einen sinnvollen Einsatz ist es aber wichtig, eine angemessene Abstraktionsebene zu wählen. Zum Beispiel ist es vollkommen unnötig, eine http-basierte Webserverattacke als Folge von Ethernet-Frames zu beschreiben. Andererseits ist die Zahl der darzustellenden Netzwerkprotokolle aber kaum überschaubar und ständig zunehmend. Somit scheint eine separate Implementation jedes Protokolls nicht besonders sinnvoll zu sein (und wäre im Rahmen dieser Arbeit auch nicht möglich).

memory Arbeitsspeicher. Auch wenn es nicht unbedingt notwendig erscheint, flüchtige Daten wie Speicherinhalte zu klassifizieren, kann dies in einigen Situationen nützlich sein. Exemplarisch sei hier Malware wie der Wurm „W32/SQLSlammer“ erwähnt, der sich nach dem Eindringen in ein System nicht auf persistenten Datenträgern verbreitet, um einer Erkennung durch Anti-Malware-Programme zu entgehen.

octectstream Bytestrom. Dies ist ein generisches Element, mit der sich jede Art von Software beschreiben lassen sollte. Allerdings sind die Identifikationsmöglichkeiten denkbar gering. Im Idealfall wird dieses Element nicht benötigt, da die spezifischeren Elemente ausreichen. Sollte ein Datenobjekt jedoch durch keines der vorherigen Elemente beschrieben werden können, ist diese allgemeine Möglichkeit vorhanden.

6.3.1.1 File

Eine Datei wird üblicherweise durch Angabe des Dateinamens und des Pfads identifiziert. Häufig (zum Beispiel in Webbrowsern) werden diese Angaben in einem URI zusammengefasst. Dies hat neben einer einheitlichen Syntax den Vorteil, auch Dateien eines anderen Systems adressieren zu können. Deshalb ist diese Alternative ebenfalls vorhanden, allerdings eingeschränkt auf die URI-Methode „file“, da die Erreichbarkeit über Netzwerkprotokolle in diesem Zusammenhang nicht notwendig ist (siehe auch Abschnittt 6.1.3).

Für Dateiname und -pfad wird der Datentyp `nonEmptyString` (also im Prinzip `xsd:string`, siehe Abschnitt 6.1.1) verwendet, da dies ein Datentyp ist, bei dem alle Zeichen unverändert erhalten bleiben[8]. Somit ist es nicht möglich, den Inhalt dieser beiden Elemente im Instanz-Dokument wie in XML üblich auf mehrere Zeilen umzubrechen, da die Zeilenumbruchzeichen dann als Teil des Dateinamens angesehen würden. Alternativ hätte der Datentyp `xsd:normalizedString` verwendet werden können, bei dem lediglich die Zeichen „`carriage return`“, „`line feed`“ und „`horizontal`

[8] Nur die Angabe eines leeren Strings ist verboten.

tab“ durch „space“ ersetzt werden. Allerdings ist selbst diese Einschränkung unzulässig, da auch diese Sonderzeichen in etlichen Dateisystemen[9] in Dateinamen gültig sind und bei der Ersetzung somit der Dateiname verändert werden würde. Die Verwendung solcher Steuerzeichen wird zwar nicht empfohlen und von vielen Programmen auch verhindert, trotzdem müssen auch diese Spezialfälle berücksichtigt werden. Die Frage der Darstellung solcher Dateinamen wird in der SCL-Spezifikation nicht geklärt, da dies als Aufgabe des interpretierenden Programms angesehen wird[10].

Aufgrund der unterschiedlichen Syntax der verschiedenen Betriebssysteme ist die Angabe eines URI zu bevorzugen. Wird stattdessen ein Dateiname- und pfad angegeben, kann nicht unbedingt davon ausgegangen werden, dass sich diese beiden Angaben zu einer gültigen Adresse zusammenfügen lassen. Bei einer Weiterverarbeitung dieser Inhalte muss also gegebenenfalls ein Pfadtrennzeichen (wie zum Beispiel „\“) zwischen Pfad und Dateiname eingefügt werden.

Die Elemente `name` und `path` dienen dazu, Dateiname und -pfad getrennt behandeln zu können. Sollte es wirklich nicht möglich sein, Pfad und Dateiname eindeutig zu trennen, ist es aber auch möglich, das `path`-Element wegzulassen und für die komplette Angabe nur das `name`-Element zu verwenden. Ebenso kann es Situationen geben, in denen keine Pfadangabe vorhanden ist (zum Beispiel bei Dateien in einem Archiv, die einzeln gemeldet werden). Auch dafür ist es sinnvoll, die Pfadangabe nur optional vorzusehen. Dadurch ergibt sich allerdings auch der Nachteil, dass Implementationen sich die Aufgabe einfach machen und nur das `name`-Element verwenden. Dies kann im Schema jedoch nicht verhindert werden (es wird allerdings in der Schema-Dokumentation davon abgeraten).

Eine alternative Lösung hätte immer die Angabe beider Elemente erfordert, dafür aber auch leere Inhalte zugelassen. Dadurch hätte sich in der späteren Verwendung aber der Nachteil ergeben, dass bei jeder Interpretation der Instanz-Dokumente zusätzlich unterschieden werden müsste, ob diese Elemente leer sind oder nicht.

Schema

```
<xsd:complexType name="fileIdentType">
  <xsd:sequence>
    <xsd:choice>
      <xsd:element name="uri" type="fileURI"/>
      <xsd:sequence>
        <xsd:element name="name" type="nonEmptyString"/>
        <xsd:element name="path" type="nonEmptyString"
          minOccurs="0"/>
      </xsd:sequence>
    </xsd:choice>
```

[9] Zum Beispiel bei dem unter Linux üblichen Dateisystem „ext2“.

[10] Eine Methode ist die Darstellung mittels der Zeichensatz-Codes solcher Zeichen. Zum Beispiel: „name mit<09>tab“

```
    <xsd:element name="mimetype" type="mimeDatatype"
      minOccurs="0"/>
    <xsd:element name="creationtime" type="xsd:dateTime"
      minOccurs="0"/>
    <xsd:element name="modificationtime" type="xsd:dateTime"
      minOccurs="0"/>
    <xsd:element name="accesstime" type="xsd:dateTime"
      minOccurs="0"/>
    <xsd:element name="length" type="xsd:nonNegativeInteger"
      minOccurs="0"/>
    <xsd:element name="checksum" type="checksumType"
      minOccurs="0" maxOccurs="99"/>
  </xsd:sequence>
</xsd:complexType>
```

Weitere hilfreiche Eigenschaften einer Datei sind die verschiedenen Zeitstempel, die Länge (in Byte) sowie die Angabe eines MIME-Typs.

Für MIME-Angaben ist ein eigener Datentyp entworfen worden, der die Syntax sowie die Hauptkategorien der MIME-Definition entsprechend einschränkt (siehe Abschnitt 6.1.10).

Für die Zeitstempel bietet sich wiederum `xsd:dateTime` an. Zwar ist nicht notwendigerweise für jede Datei eine Datums- und eine Zeitangabe zu ermitteln, dies kann aber (wie es in etlichen Dateisystemen üblichen ist) durch Einfügen von Nullwerten bzw. neutralen Werten gelöst werden. Das Jahr „`0000`“ darf nicht verwendet werden, da es im Datentyp `xsd:dateTime` (abweichend von ISO 8601) nicht zulässig ist.

Die Längenangabe umfasst bewusst auch „0“ Byte. Zwar sind solche Dateien nicht sonderlich sinnvoll und können auch keine zu klassifizierenden Inhalte enthalten. Allerdings ist die Existenz von Null-Byte-Dateien in vielen Dateisystemen möglich und für die Klassifikation kann unter Umständen bereits die Existenz einer Datei ausschlaggebend sein.

Die bisher erwähnten Angaben sind aber nicht für alle Anwendungen ausreichend, da nicht garantiert ist, dass ein Datenobjekt immer denselben Dateinamen trägt oder vielleicht durch ein identisch benanntes Datenobjekt ersetzt wurde. Auch Zeitstempel können normalerweise recht einfach manipuliert werden.

Eine eindeutige Identifikation der Datenobjekte wird in der Regel durch Prüfsummen sichergestellt[11]. Die Angabe einer Prüfsumme sowie aller weiteren Eigenschaften ist jedoch optional, da dies nicht in allen Anwendungsfällen sinnvoll ist[12] und die meisten gegenwärtigen Anti-Malware-Produkte nur den Dateinamen verwenden.

[11] Diese werden meist mittels einer Einwegfunktion wie zum Beispiel „MD5“ erzeugt.

[12] Zum Beispiel ist die aufwändige Berechnung einer Prüfsumme bei zeitkritischen Anwendungen wie einem Mailserver nicht sinnvoll.

Da zumindest theoretisch keine Prüfsumme wirklich kollisionsfrei ist, verwenden einige Integritätsprüfer mehrere verschiedene Prüfsummen-Algorithmen, um die Wahrscheinlichkeit einer Kollision zu verringern. Deshalb erlaubt das SCL-Schema hier (und auch bei jeder anderen Verwendung von Prüfsummen) mehrere Prüfsummen-Elemente (siehe Abschnitt 6.1.9). Als Obergrenze für die Anzahl der Prüfsummen wird 99 gewählt, um sowohl genügend Spielraum für zukünftige Entwicklungen zu bieten, als auch dezimal wie hexadezimal nur zwei Ziffern zu benötigen.

Instanz

```
<identification>
  <file>
    <name>ssh</name>
    <path>/usr/bin/</path>
  </file>
</identification>

<identification>
  <file>
    <uri>file://localhost/C:/malware/sample.rar</uri>
    <mimetype>application/x-rar-compressed</mimetype>
    <length>340179</length>
    <checksum type="SHA1">
      2f90bb1f9be68eecd28b715bfa2e9310de13812e
    </checksum>
  </file>
</identification>
```

6.3.1.2 Mail

Zur Beschreibung elektronischer Nachrichten (bzw. „Mail“) existieren verschiedene Standards. Während RFC 2822 (bzw. der Vorgänger RFC 822) die allgemeine Struktur festgelegt hat und sich sonst auf reine Textinhalte im ASCII-Zeichensatz beschränkt, beschreibt der MIME-Standard (RFC 2045-2049) die Behandlung anderer Inhalte und Zeichensätze.

Elektronische Nachrichten basieren aus Text-Zeilen. Jede Mail besteht aus Kopfzeilen („Header fields“), einer Leerzeile zur Trennung von Header und Body und dem eigentlichen Inhalt („Body“), der nach RFC 2822 einfach aus nicht näher festgelegten ASCII-Textzeilen besteht. Ein Header besteht wiederum aus einem Feldnamen gefolgt von einem Doppelpunkt und dem Feldinhalt.

Eine MIME-konforme Mail enthält immer bestimmte Header, die den Body näher beschreiben. Der Header „Content-Type“ gibt den MIME-Typ an, während mit „Content-

Transfer-Encoding“ eine geeignete Kodierung zur Darstellung anderer Inhalte als ASCII-Text angegeben wird. Da es abgesehen von MIME-Typen für Text-, Bild-, Audio-, Video- und anwendungsspezifische Inhalte auch die Typen „message“ und „multipart“ für die Kapselung mehrerer Inhalte bzw. kompletter Nachrichten in einem Body gibt, erlaubt dieses Konzept die Verarbeitung einer Vielzahl unterschiedlicher Nachrichten.

Hier wird auf eine vollständige Wiedergabe aller Möglichkeiten verzichtet, da nicht die originalgetreue Darstellung einer Mail, sondern nur die notwendigen Informationen zur eindeutigen Identifikation einer Mail (und gegebenenfalls einzelner Objekte innerhalb der Mail) von Interesse sind.

Schema

```
<xsd:complexType name="mailIdentType">
  <xsd:sequence>
    <xsd:element name="header" minOccurs="0"
      maxOccurs="unbounded">
      <xsd:simpleType>
        <xsd:restriction base="xsd:normalizedString">
          <xsd:pattern value="[^:]+:.*"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="mimetype" type="mimeDatatype"/>
    <xsd:element name="mimeparameter" minOccurs="0"
      maxOccurs="10">
      <xsd:simpleType>
        <xsd:restriction base="xsd:normalizedString">
          <xsd:pattern value="[^=]+=.*"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="length" type="xsd:nonNegativeInteger"
      minOccurs="0"/>
    <xsd:element name="checksum" type="checksumType"
      minOccurs="0" maxOccurs="99"/>
  </xsd:sequence>
</xsd:complexType>
```

Mit einem `mail`-Element kann nur entweder eine komplette Mail oder ein in eine MIME-Mail eingebetteter „Messagepart“ beschrieben (und somit klassifiziert) werden. Es ist aber nicht möglich, alle Bestandteile einer „message“- oder „multipart“-Mail in einer Identifikation zu beschreiben. Stattdessen lassen sich solche enthaltenen Mail-Objekte

mithilfe des `content`-Elements einzeln (siehe Abschnitt 6.3.3) darstellen. Der Unterschied besteht darin, dass dann auch jedes enthaltene Objekt einzeln klassifiziert werden muss und keine Gesamtklassifikation möglich ist.

Es können alle in der Mail bzw. dem Messagepart vorhandenen Header mittels `header` angegeben werden. Für eine eindeutige und einfache Identifikation sind jedoch vor allem die Header „Message-ID“ und „Date“ geeignet. Für MIME-konforme Mail und Messageparts ist zusätzlich der Header „Content-Transfer-Encoding“ interessant, um die Inhalte korrekt zu dekodieren. Obwohl RFC 2822 die erlaubte Syntax für Header genau vorschreibt, wird dies hier nicht umgesetzt, da in der Praxis durchaus auch illegale Header vorkommen. Um solche Header ebenfalls darstellen zu können, wird der Datentyp hier nur mittels eines regulären Ausdrucks auf die minimale Anforderung beschränkt, Feldname und Feldinhalt durch einen Doppelpunkt zu trennen. Eine weitere Eigenschaft von Mail-Headern (die in der Praxis auch oft Probleme verursacht) ist das „Folding“. Dabei kann ein Header an geeigneten Stellen auf mehrere Zeilen umgebrochen werden, um mit Beschränkungen der Zeilenlänge im SMTP-Standard konform zu sein. Da die Implementation jedoch nicht trivial ist[13] und Header in jeder Form semantisch äquivalent sind, werden hier nur „unfolded“ Header unterstützt. Einige Header sind zwar als „unstrukturiert“ definiert und können somit auch Folgen von Leerzeichen enthalten, die aufgrund des Basistyps `xsd:normalizedString` nicht erhalten bleiben. Da diese Header alleine jedoch keine eindeutige Identifikation ermöglichen, wird diese Einschränkung der Darstellungsmöglichkeiten in Kauf genommen. Da nach RFC 2822 alle Header außer „Date“ und „From“ optional sind, und diese beiden Header nicht immer korrekte bzw. aussagekräftige Inhalte besitzen, ist das `header`-Element ebenfalls optional.

Für jedes `mail`-Objekt muss ein MIME-Typ angegeben werden. Obwohl nicht jede Mail dem MIME-Standard entspricht, ist diese Voraussetzung zulässig, da gemäß RFC 2045 Abschnitt 5.2 für nicht MIME-konforme Mail der Typ „`text/plain; charset=us-ascii`“ vorausgesetzt werden darf. Da gerade MIME-Typ-Angaben bei Malware häufig gefälscht sind, muß der Inhalt des entsprechenden Headers als unzuverlässig angesehen werden und darf nicht einfach übernommen werden. Aus diesem Grund (und um konsistent mit der Verwendung bei anderen Objekten zu bleiben) wird für den MIME-Typ ein eigenes Element `mimetype` mit dem Typ `mimeDatatype` (siehe Abschnitt 6.1.10) verwendet, dessen Inhalt dem wirklichen MIME-Typ entsprechen soll. Wenn auch eine gefälschte Angabe für die Klassifikation von Bedeutung ist, kann der gefälschte MIME-Typ zusätzlich in Form eines weiteren `header`-Elements aufgezeichnet werden.

Da bei dieser Realisierung die Angabe von MIME-Typ Parametern nicht möglich ist, solche Parameter aber für die Dekodierung einer Mail durchaus wichtig sein können, wurde ein weiteres Element `mimeparameter` hinzugefügt, mit dem diese Parameter angegeben werden können. Dieses Element kann genau eine Angabe in der Form „`name=wert`“

[13] Da der Datentyp Zeilenumbrüche erhalten müsste, könnte nur `xsd:string` verwendet werden und gleichzeitig auf Zeilenumbrüche an bestimmten Stellen beschränkt werden. Dies widerspricht aber dem gewohnten Umgang mit XML-Elementinhalten.

enthalten. Mehrere Parameter können mittels mehrerer Elemente angegeben werden, wobei die maximale Anzahl RFC 2045 entsprechend nicht beschränkt ist (obwohl in der Praxis wohl nur selten mehr als zwei Parameter verwendet werden).

`checksum` und `length` haben dieselbe Funktion wie bei der `file`-Identifikation, allerdings beziehen sich beide Werte ausdrücklich nur auf den Body des beschriebenen Datenobjekts (also ohne Berücksichtigung eventuell vorhandener Header- und Trennzeilen). Eine Prüfsumme über die gesamte Nachricht inklusive Header ist nicht sinnvoll, da bei jeder Übertragung Header hinzugefügt und verändert werden können.

Das folgende Beispiel zeigt die Identifikation und Klassifikation eines infizierten Attachments in einer MIME-konformen Mail. Dabei werden nur die relevanten Teile der Mail gemeldet.

Instanz

```
<object>
  <identification>
    <mail>
      <header>From: "Infected User" user1@example.com</header>
      <header>To: user2@example.org</header>
      <header>
        Subject: I send you this file in order to have your advice
      </header>
      <header>date: Sat, 20 Oct 2001 15:42:11 +0200</header>
      <header>
        Message-ID: &lt;01234567.89ABCDEF@example.com&gt;
      </header>
      <header>MIME-Version: 1.0</header>
      <mimetype>multipart/mixed</mimetype>
      <mimeparameter>boundary="1234567890"</mimeparameter>
    </mail>
  </identification>
  <content>
    <object>
      <identification>
        <mail>
          <header>Content-Transfer-Encoding: base64</header>
          <mimetype>application/mixed</mimetype>
          <mimeparameter>name=Demo.doc.bat</mimeparameter>
          <length>157184</length>
          <checksum type="MD5">
            37a69526f514f7d9fa97f88914276f83
          </checksum>
        </mail>
      </identification>
```

```
      <classification>
        <malicious>...</malicious>
      </classification>
    </object>
  </content>
</object>
```

6.3.1.3 Sector

Mit diesem Element können einzelne Sektoren sowie Gruppen von zusammenhängenden Sektoren eines Datenträgers identifiziert werden. Der Schwerpunkt liegt dabei auf Datenobjekten, die sich nicht mittels eines Dateisystems beschreiben lassen, wie zum Beispiel Bootsektoren.

Zu jeder Sektoridentifikation muss notwendigerweise der zugrundeliegende Datenträger angegeben werden. Dazu dient das Subelement `device`. Für eine Klassifikation ist weiterhin die Sektorgröße erforderlich. Da die meisten Datenträger heutzutage eine feste Sektorgröße von 512 Byte verwenden, ist es aus Effizienzgründen allerdings wünschenswert, dies nur bei abweichenden Größen angeben zu müssen (wenn es sich zum Beispiel um Sektoren einer CDROM handelt). Deshalb wird für die Sektorgröße ein Attribut (`size`) mit dem „Default“-Wert `512` verwendet.

Aufgrund der vielen verschiedenen Arten, einen Datenträger zu beschreiben, wird für den `device`-Inhalt ein beliebiger „tokenisierter String“ erlaubt (in der Dokumentation werden aber einige sinnvolle Beispiele genannt). Demgegenüber lässt sich die Sektorgrößen-Angabe problemlos auf positive Integer-Werte beschränken.

Die weitere Struktur unterscheidet sich danach, ob das Datenobjekt nur aus einem oder aus mehreren Sektoren besteht. Da im Rahmen einer Softwareklassifikation vor allem Sektoren von Interesse sind, die ausführbaren Code enthalten bzw. am Startvorgang beteiligt sind, werden gesonderte Strukturen für den „Master Boot Record“ (als Spezialfall eines einzelnen Sektors) sowie andere Bootsektoren (als Spezialfall einer Sektorfolge) bereitgestellt.

Besteht das Datenobjekt aus mehreren Sektoren, können diese als eine oder mehrere zusammenhängende Ketten von Sektoren dargestellt werden. Jede Sektorenkette (`chain`) wird dabei durch die Adresse des Startsektors (`startaddress`) und die Adresse des Endsektors (`endaddress`) eindeutig beschrieben. Da Start- und Endadresse identisch sein können, ist auch der Fall enthalten, dass ein Datenobjekt aus einem einzelnen Sektor und einer Sektorkette besteht (da dieser Fall aber nicht sehr wahrscheinlich ist, wird auf eine eigene Struktur verzichtet). Zusätzlich muss die Anzahl aller Sektoren des Datenobjekts (also die Summe aller Kettenlängen) mittels des Elements `count` als positiver Integer-Wert angegeben werden.

Für den Spezialfall einer Kette von Bootsektoren[14] (`boot`) wird einschränkend angenommen, dass es sich immer um eine Kette von Sektoren handelt. Somit ist die Adresse des Endsektors optional, da diese aus der Sektoranzahl ermittelt werden kann.

Handelt es sich um einen einzelnen Sektor (`single`), wird nur die Adresse dieses Sektors benötigt. Das entsprechende Subelement wird trotzdem als `startaddress` bezeichnet, da es sich semantisch um denselben Inhalt wie bei Sektorketten handelt. Die Angabe einer Sektoranzahl ist dabei überflüssig und daher nicht vorgesehen.

Da für jeden bootfähigen Datenträger die Adresse des „Master Boot Records“ unveränderlich ist[15], kann bei dem entsprechenden Subelement `mbr` die Startadresse weggelassen werden. Es kann also als leeres Element `<mbr/>` verwendet werden, um die Beschreibung effizient zu halten (die Angabe einer Sektoradresse ist aber weiterhin erlaubt).

Schema

```
<xsd:complexType name="sectorIdentType">
  <xsd:sequence>
    <xsd:element name="device" type="xsd:token"/>
    <xsd:choice>
      <xsd:element name="mbr">
        <xsd:complexType>
          <xsd:sequence>
          <xsd:element name="startaddress"
            type="sectorAddressType" minOccurs="0"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="single">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="startaddress"
              type="sectorAddressType"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:sequence>
        <xsd:element name="boot">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="startaddress"
                type="sectorAddressType"/>
```

[14] Es ist nicht ausgeschlossen, dass eine Boot-Programmsequenz mehr als einen Sektor belegt.
[15] Sonst könnte die Boot-Programmsequenz nicht eindeutig vom BIOS gefunden werden.

```
            <xsd:element name="endaddress"
              type="sectorAddressType" minOccurs="0"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="count" type="xsd:positiveInteger"/>
    </xsd:sequence>
    <xsd:sequence>
      <xsd:element name="chain" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="startaddress"
              type="sectorAddressType"/>
            <xsd:element name="endaddress"
              type="sectorAddressType"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="count" type="xsd:positiveInteger"/>
    </xsd:sequence>
  </xsd:choice>
  <xsd:element name="description" type="xsd:token"
    minOccurs="0"/>
  <xsd:element name="checksum" type="checksumType"
    minOccurs="0" maxOccurs="99"/>
</xsd:sequence>
<xsd:attribute name="size" type="xsd:positiveInteger"
  default="512"/>
</xsd:complexType>
```

Bevor die Adressangabe näher erläutert wird, sei noch erwähnt, dass wie bei den vorherigen Identifikations-Elementen wieder Checksummen angegeben werden können. Für Sektor-Elemente ist es darüber hinaus möglich, mittels des Elements `description` eine informelle Beschreibung des Datenobjekts anzugeben, um zum Beispiel bekannte Datenobjekte auf Sektorebene zu benennen (etwa „Boot Parameter Block“ oder „File Allocation Table“).

Die Angabe einer Sektoradresse erfolgt mittels eines eigenen Datentyps, der verschiedene Adressierungmöglichkeiten bereitstellt.

Schema

```
<xsd:complexType name="sectorAddressType">
  <xsd:sequence>
    <xsd:element name="lba" type="xsd:nonNegativeInteger"/>
```

```
      <xsd:element name="chs" minOccurs="0">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="diskCylinder"
              type="xsd:unsignedInt"/>
            <xsd:element name="diskHead"
              type="xsd:unsignedByte"/>
            <xsd:element name="diskSector"
              type="xsd:unsignedShort"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
```

Bei der „Logical Block Address“ handelt es sich einfach um einen Integer-Wert (beginnend bei 0), der sich daher sehr einfach implementieren lässt. Die vor allem bei älteren IDE-Festplatten verwendete „C/H/S“-Methode lässt sich dagegen am sinnvollsten durch drei einzelne Elemente darstellen. Dies ist zwar nicht besonders effizient, ein zusammengesetzter Inhalt hätte aber den Nachteil, die semantische Bedeutung einer numerischen Angabe zu verlieren.

Instanz

```
<object>
  <identification>
    <sector>
      <device>hda</device>
      <mbr/>
    </sector>
  </identification>
  <classification>...</classification>
</object>

<object>
  <identification>
    <sector>
      <device>multi(0)disk(0)rdisk(0)partition(1)</device>
      <chain>
        <startaddress><lba>70</lba></startaddress>
        <endaddress><lba>112</lba></endaddress>
      </chain>
      <count>43</count>
      <description>File Allocation Table</description>
```

```
    </sector>
  </identification>
  <classification>...</classification>
</object>
```

6.3.1.4 Packet

Einerseits soll es mit diesem Element ermöglicht werden, Malware bereits während der Übertragung im Netzwerk zu klassifizieren. Da solche Datenobjekte typischerweise größer als ein einzelnes Netzwerkpaket sind, muss also eine zusammenhängende Folge von Netzwerkpaketen dargestellt werden können. Anderseits sind auch die Anforderungen von Anwendungen wie „Intrusion Detection Systems“ oder Portscan-Detektoren zu berücksichtigen, die unter Umständen auch aus einem scheinbar zufälligen „Muster“ einzelner Pakete einen Zusammenhang erkennen können. Deshalb muss auch die Möglichkeit bestehen, einzelne Pakete sowie Gruppen unabhängiger Pakete (zum Beispiel eine Gruppe einzelner Pakete an verschiedenen Ports in bestimmten zeitlichen Abständen) im Rahmen einer Klassifikation zu beschreiben.

Schema

```
<xsd:complexType name="packetIdentType">
  <xsd:sequence>
    <xsd:choice maxOccurs="unbounded">
      <xsd:element name="single">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:group ref="packetStruct"/>
            <xsd:element name="checksum" type="checksumType"
              minOccurs="0" maxOccurs="99"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="sequence">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:group ref="packetStruct"/>
            <xsd:element name="count"
              type="xsd:positiveInteger">
            <xsd:element name="checksum" type="checksumType"
              minOccurs="0" maxOccurs="99"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
```

```
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

Ein `packet`-Element enthält somit eine Folge beliebig vieler `single`- und `sequence`[16]-Elemente, die jeweils ein einzelnes Paket bzw. eine Folge zusammenhängender Pakete beschreiben. Auf diese Weise lassen sich alle denkbaren Paketkombinationen darstellen. Da Einzelpakete und Paketfolgen auf nahezu identische Weise beschrieben werden können, wird die Paketstruktur in einer Elementgruppe definiert, auf der sowohl `single` wie `sequence` aufbauen. Für eine Folge von Paketen ist nur die zusätzliche Angabe der Paketanzahl mittels des Elements `count` erforderlich.

Im Sinne einer optimalen Modularisierung hätte auch das `checksum`-Element in die `paketStruct`-Gruppe eingefügt werden können. Allerdings wurde die Konsistenz mit den anderen Objekt-Identifikationen als wichtiger erachtet, bei denen `checksum` ebenfalls immer das letzte Element innerhalb der Identifikation darstellt[17].

Schema

```
<xsd:group name="packetStruct">
  <xsd:sequence>
    <xsd:element name="protocol" type="xsd:NMTOKEN"/>
    <xsd:element name="source" type="packetAddressType"
      minOccurs="0"/>
    <xsd:element name="destination" type="packetAddressType"
      minOccurs="0"/>
    <xsd:element name="time" type="xsd:dateTime" minOccurs="0">
    <xsd:element name="length" type="xsd:positiveInteger">
      minOccurs="0"/>
  </xsd:sequence>
</xsd:group>
```

Die Beschreibung der Paketstruktur umfasst das Protokoll, Quell- und Zieladresse, Empfangs- bzw. Beobachtungszeitstempel sowie die Länge des Pakets bzw. der Paketfolge (in Byte).

Wie bereits erwähnt, wird die Menge der darstellbaren Protokolle nicht beschränkt. Deshalb könnte für die Protokollbezeichnung im Prinzip ein Datentyp `xsd:string` oder `xsd:token` verwendet werden. Da aber eine einheitliche Angabe sinnvoll ist und Protokollnamen oft in Verbindung mit anderen syntaktischen Beschränkungen verwendet werden (zum Beispiel in URIs), ist eine Beschränkung der verwendbaren „Interpunktionszeichen“ möglich und sinnvoll. Es wurde der vordefinierte Typ `xsd:NMTOKEN` gewählt,

[16] Nicht zu verwechseln mit dem Element `xsd:sequence` des XML Schema Standards.

[17] Da in den Instanz-Dokumenten bis zu 99 Prüfsummen in Folge erlaubt sind, ist es auch sinnvoller, diese nicht mitten zwischen anderen Identifikationsdaten zu platzieren.

um insbesondere Leerzeichen und Anführungszeichen auszuschließen. Als sinnvolle Protokollnamen werden Angaben wie zum Beispiel „http“, „dns“, „icmp“ oder „ipv6“ angesehen, die auch in den Netzwerkanwendungen selber verwendet werden und meist an zentraler Stelle (zum Beispiel /etc/protocols und /etc/services unter Unix) festgelegt werden.

Für den Zeitstempel ist der vordefinierte Datentyp `xsd:dateTime` ausreichend, da dieser auch die Angabe von Sekundenbruchteilen erlaubt. Der Zeitstempel bezieht sich bei einer Folge von Paketen auf das erste Paket. Ist die Ankunftszeit jedes Pakets wichtig, können diese nicht als `sequence` sondern nur als Folge von `single`-Elementen dargestellt werden.

Aufgrund der Vielzahl unterschiedlicher Adressierungsarten wird für Quell- und Zieladresse wiederum ein eigener Datentyp verwendet, der eine Auswahl aus etlichen Möglichkeiten zulässt.

Schema

```
<xsd:complexType name="packetAddressType">
  <xsd:sequence>
    <xsd:element name="interface" minOccurs="0">
      <xsd:complexType>
        <xsd:choice>
          <xsd:element name="mac" type="macAddress"/>
          <xsd:element name="eui64" type="eui64Address"/>
          <xsd:element name="other" type="xsd:token"/>
        </xsd:choice>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="host">
      <xsd:complexType>
        <xsd:choice>
          <xsd:element name="ip" type="ipAddress"/>
          <xsd:element name="ipv6" type="ipv6Address"/>
          <xsd:element name="dnsname" type="xsd:NMTOKEN"/>
          <xsd:element name="winsname" type="xsd:token"/>
          <xsd:element name="other" type="xsd:token"/>
        </xsd:choice>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="service" type="xsd:NMTOKEN"
      minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

Eine Adresse kann also durch Angabe von `interface`, `host` und `service` bestimmt werden (wobei nur das `host`-Element notwendig ist).

`service` soll eine Servicenummer (zum Beispiel die Portnummer einer TCP-Verbindung oder den ICMP-Nachrichtentyp) oder einen Servicenamen enthalten. Für die `interface`-Angabe existieren Subelemente für MAC- und EUI64-Adressen, die sowohl als ein hexadezimaler Wert wie auch in einem typischen Format angegeben werden dürfen (siehe Abschnitt 6.1.5 und 6.1.6). Zusätzlich existiert mit `other` die Möglichkeit, weitere Interface-Adressarten darzustellen, solange diese mittels `xsd:token` beschrieben werden können.

Für `host` wurden mehrere verbreitete Möglichkeiten vorgesehen. Während mit den Elementen `ip` und `ipv6` ausschließlich gültige IP-Adressen angegeben werden können (siehe Abschnitt 6.1.7 und 6.1.8 für die entsprechenden Datentyp-Definitionen), wird für die Elemente `dnsname` und `winsname` nur die semantische Bedeutung, dass es sich um einen DNS[18]- bzw. WINS[19]-Rechnernamen handeln soll, festgelegt. Auch hier existiert mit dem Element `other` wieder eine weitere generische Identifikationsmöglichkeit.

Instanz

```
<object>
  <identification>
    <packet>
      <sequence>
        <protocol>tcp</protocol>
        <source>
          <interface><mac>12:34:56:78:9a:bc</mac></interface>
          <host><ip>192.168.1.1</ip></host>
          <service>32776</service>
        </source>
        <destination>
          <interface>
            <eui64>12-34-56-ff-ff-cb-a9-87</eui64>
          </interface>
          <host><ip>192.168.1.2</ip></host>
          <service>631</service>
        </destination>
        <length>918</length>
        <count>10</count>
        <checksum type="MD5">
          6f2a464628809e9d896793892d589c9d
        </checksum>
```

[18] „Domain Name System“
[19] „Windows Internet Name Service“

```
      </sequence>
    </packet>
  </identification>
  <classification>...</classification>
</object>
```

6.3.1.5 Memory

Mit diesem Element können Datenobjekte identifiziert werden, die sich in einem flüchtigen Speicher befinden. Dies kann notwendig sein, um festzustellen, ob eine bestimmte Art von Malware zurzeit aktiv ist. Darüber hinaus existiert auch Malware, die ausschließlich im flüchtigen Speicher zu finden ist. Naturgemäß veralten Informationen über solche Datenobjekte schnell und eignen sich nicht für Anwendungen, die den Zustand eines Systems über einen längeren Zeitraum protokollieren.

Aufgrund der flüchtigen Natur ist die genaue Lokalisierung weniger bedeutend als bei persistenten Datenobjekten. In der Regel ist es ausreichend, die Existenz eines bestimmten Datenobjekts feststellen zu können.

Schema

```
  <xsd:complexType name="memoryIdentType">
    <xsd:sequence>
      <xsd:element name="startaddress" type="xsd:token">
      </xsd:element>
      <xsd:element name="endaddress" type="xsd:token"
        minOccurs="0"/>
      <xsd:element name="length" type="xsd:positiveInteger">
      </xsd:element>
      <xsd:element name="description" type="xsd:token"
        minOccurs="0"/>
      <xsd:element name="checksum" type="checksumType"
        minOccurs="0" maxOccurs="99"/>
    </xsd:sequence>
  </xsd:complexType>
```

Die einzig notwendige Angabe ist somit die Größe des Datenobjekts, obwohl die Angabe einer Prüfsumme ebenfalls sehr empfehlenswert ist.

Sollte die Lokalisierung des Datenobjekts von Interesse sein (etwa um Malware während der Laufzeit aus dem Speicher entfernen zu können), kann dies durch Angabe einer Start- und Endadresse erfolgen. Das Format der Speicheradressen ist nicht weiter eingeschränkt, da durch die Vielzahl verschiedener Speicheradressierungsmodelle eine einheitliche Angabe kaum möglich erscheint.

6.3.1.6 Octetstream

Dieses Element existiert nur, um auch Datenobjekte behandeln zu können, die sich mit keiner anderen Kategorie beschreiben lassen. In diesem Sinne handelt es sich um eine minimale Beschreibung eines Datenobjekts.

Dazu wird die grundlegende Annahme getroffen, dass jedes von einem Computersystem verarbeitbare Datenobjekt durch eine Folge bzw. einem Strom von Bytes repräsentiert werden kann. Weiterhin wird davon ausgegangen, dass jedes klassifizierbare Datenobjekt eine endliche Größe besitzt, die zwar nicht von vorne herein bekannt sein muss, aber durch Lesen des vollständigen Datenobjekts ermittelt werden kann.

Schema

```
<xsd:complexType name="streamIdentType">
  <xsd:sequence>
    <xsd:element name="source" type="xsd:token"
      minOccurs="0"/>
    <xsd:element name="destination" type="xsd:token"
      minOccurs="0"/>
    <xsd:element name="mimetype" type="mimeDatatype"
      minOccurs="0"/>
    <xsd:element name="description" type="xsd:token"
      minOccurs="0"/>
    <xsd:element name="length" type="xsd:positiveInteger"/>
    <xsd:element name="checksum" type="checksumType"
      maxOccurs="99"/>
  </xsd:sequence>
</xsd:complexType>
```

Der Endlichkeitsvoraussetzung entsprechend muss die Größe des Datenobjekts angegeben werden. Zusätzlich wird die Angabe wenigstens einer Prüfsumme gefordert, da die Länge allein keine eindeutige Klassifikation ermöglicht.

Schließlich können (falls bekannt) Quelle, Ziel, MIME-Typ und sonstige Beschreibungen ohne zusätzliche Einschränkungen angegeben werden.

6.3.2 Klassifikation

Das `classification`-Element ordnet ein Datenobjekt einer der in Abschnitt 3.2.2 bzw. 4.4 beschriebenen Softwareklassen `malicious`, `verified`, `unknown`, `modified` oder `update` zu.

Schema

```
<xsd:complexType name="classificationType">
  <xsd:choice>
    <xsd:element name="malicious" type="malClassType"
      maxOccurs="unbounded"/>
    <xsd:element name="verified" type="goodClassType"/>
    <xsd:element name="unknown" type="unknownClassType"/>
    <xsd:element name="modified" type="modifiedClassType"/>
    <xsd:element name="outdated" type="updateClassType"/>
  </xsd:choice>
</xsd:complexType>
```

Die Klasse `malicious` dient zur Kennzeichnung von Malware, während die Klasse `verified` „harmlose“ bzw. vertrauenswürdige Datenobjekte als solche auszeichnen soll.

Datenobjekte, deren Zustand sich (gegenüber einem mit einer geeigneten Methode festgehaltenen Systemzustand) verändert hat, werden von der Klasse `modified` berücksichtigt. Dies beinhaltet keine Aussage über die Gut- oder Bösartigkeit des Datenobjekts und könnte somit auch zusätzlich zu einer Einordnung als `verified` oder `malicious` erfolgen (obwohl das Letztere wohl nicht viel Sinn ergibt).

Auf ähnliche Weise kann auch die Klasse `update` verwendet werden. Allerdings erfolgt hier kein Vergleich auf eine (eventuell unbefugte) Zustandsänderung, sondern auf eine nicht erfolgte aber erwünschte Zustandsänderung. Dies dient zur Erfassung typischer „Update“-Situationen.

6.3.2.1 Malicious

Die Klassifikation von Malware erfolgt in der Regel durch Angabe eines Malware-Namens und der Erkennungsmethode. Ein Malware-Name kann wiederum auf verschiedene Weise angegeben werden. Einerseits kann der Name als einzelner String gemäß den CARO-Konventionen angegeben werden (Element `caroname`), andererseits kann für jeden Namensbestandteil ein eigenes Subelement verwendet werden.

Um auch die Konvertierung von anderen Report-Formaten zu vereinfachen (dies wird in der Testauswertung des aVTC häufig benötigt), ist es zusätzlich möglich, nur ein einzelnes Element (`mwname`) ohne weitere Vorgaben zu verwenden. Diese Möglichkeit hat allerdings den großen Nachteil, dass die syntaktischen Strukturen der Malware-Klassifikation im Elementinhalt verborgen sind und somit alle Vorteile der XML-Struktur ungenutzt bleiben. Trotzdem hat auch diese Möglichkeit noch Vorteile gegenüber anderen Lösungen, da weiterhin eine eindeutige Trennung der Objekt-Identifikation und -Klassifikation gegeben ist.

In diesem Sinne sind die anderen Strukturen dieser Klasse eher als Möglichkeit zusätzlicher Vereinheitlichung anzusehen statt als Zwang zu einer festgelegten Struktur.

Alternativ zu Malware-Namen ist es auch möglich, mittels `id` eine ID anzugeben, die auf einen Eintrag in einer Datenbank verweist. Dieses Verfahren wird heutzutage zwar nicht bei Anti-Malware-Produkten verwendet, aber unter anderem zur Kennzeichnung von Verwundbarkeiten bestimmter Software (beispielsweise die „Bugtraq“-Datenbank[20] oder das CVE[21]-Projekt). Da dabei zum Teil nicht nur die Softwareschwächen selber, sondern auch die konkrete Ausnutzung(„Exploit“) dieser Schwächen erfasst werden (und diese durchaus als Malware klassifiziert werden könnten), sollte diese Möglichkeit hier ebenfalls berücksichtigt werden.

Bei allen diesen Möglichkeiten können (auch mehrere) zusätzliche Beschreibungen in Form von `property`-Elementen angehängt werden.

Um diese Struktur zu realisieren, wird deshalb innerhalb einer Elementfolge (`xsd:sequence`) eine Auswahl (`xsd:choice`) zwischen den verschiedenen Alternativen (`id`, `caroname`, `mwname` bzw. zusammengesetzte Malware-Namen) deklariert. Die Elementfolge enthält danach nur noch das `property`-Element, wobei dieses beliebig oft vorkommen, aber auch ganz fehlen darf.

Schema

```
<xsd:complexType name="malClassType">
  <xsd:sequence>
    <xsd:choice>
      <xsd:element name="id">
        <xsd:complexType>
          <xsd:simpleContent>
            <xsd:extension base="xsd:token">
              <xsd:attribute name="type" type="xsd:token"
                use="required"/>
            </xsd:extension>
          </xsd:simpleContent>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="caroname" type="caroNameType"/>
      <xsd:sequence>
        <xsd:element name="mwtype" type="noSpaceToken"
          minOccurs="0"/>
        <xsd:element name="mwplatform" type="noSpaceToken"
          minOccurs="0"/>
        <xsd:element name="mwname" type="xsd:token"/>
          <xsd:annotation>
```

[20] http://www.securityfocus.com/bid

[21] „Common Vulnerabilities and Exposures“, http://cve.mitre.org/

```
            <xsd:documentation>
              "unknown" SHOULD be used as name for any malware
              that can't be identified by a more exact name.
            </xsd:documentation>
          </xsd:annotation>
          <xsd:element name="mwvariant" type="noSpaceToken"
            minOccurs="0"/>
          <xsd:element name="mwmodifier" type="noSpaceToken"
            minOccurs="0"/>
        </xsd:sequence>
      </xsd:choice>
      <xsd:element name="property" type="xsd:token"
        minOccurs="0" maxOccurs="unbounded">
    </xsd:sequence>
    <xsd:attribute name="method" type="mwMethodType"
      default="exact identification"/>
</xsd:complexType>
```

Die Inhalte von `id` und `property` werden dabei nicht weiter beschränkt (abgesehen von der „whitespace“-Ersetzung). `id` erfordert aber zusätzlich die Angabe des Attributs `type`, das die verwendete Datenbank identifiziert.

Um die Angabe der Malware-Namen wirklich flexibel zu gestalten, sind in dem Element `mwname` auch Leerzeichen erlaubt, da dies von einigen Produkten in deren spezifischer Syntax verwendet wird. Für die anderen Namensbestandteile ist dies nicht notwendig, da deren Inhalte zwar nur informell aber doch ausreichend genau vorgegeben sind, um die Notwendigkeit von Leerzeichen zu vermeiden. Für Vergleichszwecke (wie im aVTC) ist es ohnehin unproblematisch, Leerzeichen bei einer Konvertierung zu ersetzen (typischerweise durch Unterstriche).

Die einzelnen Namensbestandteile `mwtype`, `mwplatform`, `mwvariant`, `mwmodifier` und `mwname` (wenn es in Verbindung mit den anderen Elementen verwendet wird) entsprechen weitgehend der Bedeutung der Namensbestandteile der CARO-Konvention, allerdings ohne die vorgegebene Syntax (diese wird durch die XML-Struktur ersetzt) und ohne Aufzählungen fest vorgegebener Werte (es werden in der Schema-Dokumentation aber Empfehlungen gegeben).

Instanz

```
  <classification>
    <malicious>
      <mwtype>worm</mwtype>
      <mwplatform>W32</mwplatform>
      <mwname>Sircam</mwname>
      <mwmodifier>@MM</mwmodifier>
```

```
    </malicious>
  </classification>
```

Zusätzlich wird für das Element `mwname` festgelegt, dass der Inhalt „`unknown`“ eine besondere Bedeutung hat (nämlich als Kennzeichnung für unbekannte Namen, wie dies bei heuristischer Erkennung häufig der Fall ist) und nicht als Name für ein bestimmtes Malware-Objekt zu interpretieren ist. Dies lässt sich leider nur mittels der Schema-Dokumentation beschreiben und somit durch Validierung mittels des SCL-Schemas nicht überprüfen. Es gibt also keine Möglichkeit, interpretierende Anwendungen dazu zu „zwingen“, dies zu berücksichtigen.

Für das Element `caroname` wird die fest vorgegebene Syntax der CARO-Konvention (siehe Abschnitt 3.1.2) gefordert.

Dazu lassen sich die Namensbestandteile `malwaretype`, `platform`, `family_name`, `group_name` und `packer` je durch folgenden regulären Ausdruck beschreiben:

```
[-_A-Za-z0-9]{1,20}
```

Dieser Ausdruck wird im Folgenden durch das Zeichen □ abgekürzt.

Die Bestandteile `infective_length`, `subvariant` und `devolution` lassen sich darüber hinaus noch auf Ziffern bzw. Buchstaben einschränken. Zusätzlich ist zu berücksichtigen, dass nur entweder `infective_length` oder `subvariant` angegeben werden darf und die Angabe der `devolution` weggelassen werden kann. Dies lässt sich durch diesen Ausdruck realisieren:

```
([0-9]{1,20}|[A-Za-z]{1,20}[0-9]{0,20})
```

Die Angabe des Sprachcodes lässt sich (inklusive aller Groß- und Kleinschreibungen) ausdrücken durch:

```
([A-Za-z]{2}|[Uu][Nn][Ii])
```

Der „mailing“-Anhang `@M` und der „mass mailing“-Anhang `@MM` lassen sich durch folgenden Ausdruck in allen erlaubten Varianten darstellen:

```
@[Mm][Mm]?
```

Schließlich kann noch ein Hersteller-spezifischer Namensteil angehängt werden, der keinen weiteren Einschränkungen unterliegt und sich also durch `.*` darstellen lässt.

Alle Bestandteile von `infective_length` bis zum Hersteller-spezifischen Teil lassen sich dabei inklusive der Trennzeichen durch diesen regulären Ausdruck realisieren (der im Folgenden durch das Zeichen ● abgekürzt wird):

Schema

```
([0-9]{1,20}|[A-Za-z]{1,20}[0-9]{0,20})((:([A-Za-z]{2}|[Uu]
[Nn][Ii]))?(#□)?(@[Mm][Mm]?)?(!.*)?)?
```

Somit lässt sich der gesamte reguläre Ausdruck für gültige CARO-Namen einigermaßen übersichtlich darstellen:

Schema

```
 <xsd:simpleType name="caroNameType">
  <xsd:restriction base="noSpaceToken">
      <xsd:pattern value="□://□/□(\.□)?\.●"/>
      <xsd:pattern value="□://\{□(,□)+\}/□(\.□)?\.●"/>
      <xsd:pattern value="\{□(,□)+\}://□/□(\.□)?\.●"/>
      <xsd:pattern value="\{□(,□)+\}://\{□(,□)+\}/□(\.□)?\.●"/>
  </xsd:restriction>
</xsd:simpleType>
```

Das zweite Muster unterscheidet sich von dem ersten dabei nur dadurch, dass es auch die Angabe mehrerer Malware-Plattformen erlaubt (zum Beispiel „`virus://{W97M,X97M,A97M}/`“). Dementsprechend erlauben die letzten beiden Muster auch mehrere Malware-Typen. Dies hätte sich zwar auch durch ein einziges Muster realisieren lassen, aus Gründen der Übersichtlichkeit wurden jedoch mehrere kürzere Muster verwendet.

Da auch die CARO-Konvention von Zeit zu Zeit aktualisiert wird, um neue Malware-Typen und -Plattformen zu berücksichtigen, werden diese Angaben hier nicht auf eine Liste erlaubter Werte beschränkt.

Instanz

```
  <classification>
    <malicious>
      <caroname>virus://Word97Macro/Melissa.A@mm</caroname>
    </malicious>
  </classification>
```

Die Erkennungsmethode wird aus Effizienzgründen als Attribut zum Element `malicious` hinzugefügt. Da zumindest für Anti-Malware-Produkte die überwiegende Mehrzahl aller Klassifikationen auf Übereinstimmung mit einem Suchmuster basiert, lässt sich durch Angabe des „Default“-Wertes `“exact identification”` im Schema die Angabe der Methode in den meisten Fällen vermeiden[22].

Schema

```
<xsd:simpleType name="mwMethodType">
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="exact identification"/>
    <xsd:enumeration value="generic identification"/>
```

[22] Für Elemente kann kein Standardwert vorgegeben werden.

```
    <xsd:enumeration value="heuristic detection"/>
    <xsd:enumeration value="other"/>
  </xsd:restriction>
</xsd:simpleType>
```

Außer der „exakten“ Erkennung ist die in Abschnitt 3.1.1 beschriebene „generische“ und „heuristische“ Erkennung berücksichtigt worden. Mit „`other`“ ist auch dieser Datentyp für andere bzw. zukünftige Anwendungen nutzbar.

Instanz

```
  <classification>
    <malicious method="heuristic detection">
      <mwname>unknown</mwname>
    </malicious>
  </classification>
```

6.3.2.2 Verified

Da diese Klasse heutzutage noch nicht zur Software-Klassifikation verwendet wird, ist es schwierig, eine genaue Syntax vorzugeben, ohne dabei die zukünftige Verwendbarkeit einzuschränken. Deshalb werden hier nur wenige, möglichst offene Strukturen vorgegeben, die eventuell später überarbeitet oder ergänzt werden müssen.

Schema

```
<xsd:complexType name="goodClassType">
  <xsd:sequence>
    <xsd:element name="type" type="xsd:token" minOccurs="0"/>
    <xsd:element name="name" type="xsd:token" minOccurs="0"/>
    <xsd:element name="method" type="xsd:token"/>
    <xsd:element name="trustbase" type="xsd:token"/>
  </xsd:sequence>
</xsd:complexType>
```

Mit `method` wird die Methode angegeben, mit der die Integrität des Datenobjekts überprüft wurde (zum Beispiel eine kryptographische Signatur oder eine Prüfsumme).

Das Element `trustbase` identifiziert die Datenbasis, die die Bindung des Datenobjekts an den Hersteller bzw. an das Computersystem herstellt. Dabei könnte es sich um eine vom Hersteller bereitgestellte Liste von Systemkomponenten (inklusive Integritätsdaten) handeln. Beide Angaben sind notwendig, um die Zuverlässigkeit der Klassifikation einschätzen zu können.

Die Angabe dieser beiden Elemente bei jedem Datenobjekt bedeutet natürlich eine deutlich erhöhte Datenmenge, die unter Umständen einen hohen redundanten Anteil aufweist. Während sich dieses Problem für die `malicious`-Klassifikation durch Verwendung des `dataversion`-Elements im `origin`-Abschnitt immer lösen lässt, ist dies für eine Verifikation nicht unbedingt möglich, da die Datenbasis nicht unbedingt vom überprüfenden Programm abhängig ist. Zusätzlich ist es bisher nicht möglich, eine Standardmethode vorzugeben, die in der Mehrzahl der Fälle angewendet wird. Trotz dieser Nachteile sind diese Angaben nicht optional, da die Klassifikation sonst den Großteil ihrer Aussagekraft verlieren würde.

Für die im Folgenden beschriebenen Klassen `modified` und `update` stellt sich dieses Redundanzproblem im Prinzip zwar auch, allerdings kann dort davon ausgegangen werden, dass auf einem System jeweils nur relativ wenige[23] Datenobjekte verändert wurden bzw. einer Veränderung bedürfen. Sollte diese Annahme nicht zutreffen, stellt sich ohnehin die Frage, ob eine dermaßen umfangreiche Logdatei noch ihren Sinn erfüllt[24]. Somit kann ein gewisser redundanter Anteil dort durchaus akzeptiert werden.

Die optionalen Elemente `type` und `name` sollen eine Möglichkeit bieten, zusätzliche Informationen zur Bedeutung des Datenobjekts innerhalb des Computersystems angeben zu können.

Entsprechend dem Ziel der breiten Anwendbarkeit (und mangels praktischer Erfahrungen mit dieser Art von Klassifikation) wird für alle Elemente der Datentyp `xsd:token` verwendet.

Instanz

```
<object>
  <identification>
    <file>
      <name>bash</name>
      <path>/bin/</path>
      <checksum type="MD5">
        9b79e3d424f33dee7801de6d968dbe30
      </checksum>
    </file>
  </identification>
  <classification>
    <verified>
      <type>Command interpreter</type>
      <name>GNU Bourne-Again Shell</name>
      <method>checksum: MD5</method>
      <trustbase>local RPM database</trustbase>
```

[23] Im Vergleich zu der Gesamtheit aller auf einem System vorhandenen Datenobjekte, die potentiell alle in den Klassen `verified` und `unknown` enthalten sein könnten.

[24] Nämlich, einen Überblick über die wesentlichen sicherheitsrelevanten Veränderungen zu bieten.

```
    </verified>
  </classification>
</object>
```

6.3.2.3 Unknown

Diese Klasse umfasst alle Datenobjekte, die nicht eindeutig einer der anderen Klassen zugeordnet werden können. Deshalb ist die Struktur des `unknown`-Elements auch sehr einfach gestaltet. Es gibt nur ein einziges Subelement `property`, das allerdings unbegrenzt oft (oder auch gar nicht) vorkommen darf.

Schema

```
<xsd:complexType name="unknownClassType">
  <xsd:sequence>
    <xsd:element name="property" minOccurs="0"
      maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:simpleContent>
          <xsd:extension base="xsd:token">
            <xsd:attribute name="reason" use="optional"/>
          </xsd:extension>
        </xsd:simpleContent>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

`property` kann beliebige Texte enthalten und dient zur Angabe von weiteren Objekteigenschaften, die für eine Bewertung hilfreich sein könnten.

Instanz

```
  <classification>
    <unknown>
      <property>may be a corrupted file</property>
      <property>
        contains inactive malicious code fragments
      </property>
    </unknown>
  </classification>
```

Mittels der Inline-Dokumentation werden zwei Inhalten besondere Bedeutungen zugewiesen:

„`not infected`“ dient dazu, für Datenobjekte explizit angeben zu können, dass sie nicht mit replizierender Malware infiziert sind (dies muss natürlich durch die Analyse des Datenobjekts erwiesen sein). Diese Möglichkeit existiert in Anlehnung an die Report-Formate vieler Anti-Malware-Produkte, die (oft nur mit zusätzlichen Optionen) zu jeder Datei eine Status-Meldung ausgeben und solche Dateien dann meist als „OK“ kennzeichnen.

„`not analysed`“ besagt, dass das Datenobjekt nicht analysiert werden konnte und eine Klassifikation somit nicht möglich ist. In diesem Fall sollte unbedingt zusätzlich das Attribut `reason` verwendet werden, das dafür vorgesehen ist, die Ursache für solche Situationen wiederzugeben[25].

Instanz

```
<classification>
  <unknown>
    <property reason="compression format not supported">
      not analysed
    </property>
  </unknown>
</classification>
```

6.3.2.4 Modified

Während die Klassen `malicious` und `unknown` sich eher an den Anforderungen von Anti-Malware-Programmen orientieren, ist diese Klasse mehr für Integritäts-Prüfprogramme gedacht. Somit ist die Verwendung von `modified` auch eher als Gegensatz zur Klasse `verified` sinnvoll.

Mit dieser Klasse können Objekt-Eigenschaften, die von einem erwarteten Zustand abweichen, dargestellt werden.

Schema

```
<xsd:complexType name="modifiedClassType">
  <xsd:sequence>
    <xsd:sequence maxOccurs="unbounded">
      <xsd:element name="found" type="xsd:token"/>
      <xsd:element name="expected" type="xsd:token"/>
    </xsd:sequence>
```

[25] Aufgrund der Definition kann dieses Attribut für jedes `property`-Element verwendet werden. Dies kann zwar auch bei anderen Meldungen nützlich sein, es wird jedoch nur für solche Fehlersituationen als notwendig angesehen.

```
    <xsd:element name="method" type="xsd:token"/>
    <xsd:element name="base" type="xsd:token"/>
  </xsd:sequence>
</xsd:complexType>
```

Dazu braucht für jede Abweichung lediglich eine Angabe des vorgefundenen Zustands (Element `found`) und des erwarteten Zustands (Element `expected`) angegeben zu werden. Es können beliebig viele Eigenschaften eines Objekts angegeben werden (aber mindestens eine).

Außerdem muss für jedes Objekt die Methode (Element `method`) und eine Identifikation der Datenbasis, auf der der Vergleich beruht (Element `base`), angegeben werden.

Instanz

```
  <classification>
    <modified>
      <found>mtime: 2004-02-25 15:56</found>
      <expected>mtime: 2003-09-23 19:03</expected>
      <found>md5: 1f62a7af07af53d5fed8d6a7f0b5879b</found>
      <expected>md5: 852d7ca0f51415a4ee39bf90eaadb49a</expected>
      <method>checksum</method>
      <base>local database: /media/cdrom/tw.db</base>
    </modified>
  </classification>
```

6.3.2.5 Update

Diese Klasse ist ähnlich einfach umgesetzt worden wie die Klasse `modified`.

Auch hier wird ein „Ist“-Zustand mit einem „Soll“-Zustand verglichen. Allerdings werden andere Elementnamen verwendet, um die unterschiedliche Bedeutung hervorzuheben[26].

Schema

```
<xsd:complexType name="updateClassType">
  <xsd:sequence>
    <xsd:element name="current" type="xsd:token"/>
    <xsd:element name="new" type="xsd:token"/>
    <xsd:element name="location" type="xsd:anyURI" minOccurs="0"/>
    <xsd:element name="method" type="xsd:token"/>
```

[26] Während eine Zustandsänderung im Rahmen einer Integritätsprüfung meist unerwünscht ist, ist bei einer Suche nach veralteten Objekten das Gegenteil der Fall.

```
    <xsd:element name="base" type="xsd:token"/>
  </xsd:sequence>
</xsd:complexType>
```

Das Element `current` enthält den momentanen Zustand eines Datenobjekts, während das Element `new` den empfohlenen Zustand dieses Objekts enthält. „Empfohlen“ bedeutet dabei, dass es sich um die Behebung einer bekannten Sicherheitslücke oder eines anderen Fehlers handelt. Es muss sich dabei aber nicht um eine unabwendbare Notwendigkeit handeln, da ein „Update“ immer auch ein Risiko für die Funktionalität eines Systems darstellt und nicht jedes Software-Update sicherheitsrelevant ist.

Im Gegensatz zu einer Integritätsprüfung ist es hier ausreichend, ein eindeutiges Merkmal für den momentanen Zustand heranzuziehen (zum Beispiel die Versionsnummer einer Systemkomponente).

Die Elemente `method` und `base` haben dieselbe Bedeutung und Syntax wie in der Klasse `modified`. Zusätzlich kann hier aber mittels `location` eine URI angegeben werden, von der die neue Version des Datenobjekts erhältlich ist.

Instanz

```
  <classification>
    <update>
      <current>3.7.2-36</current>
      <new>3.7.2-72</new>
      <location>
        ftp://vendor.example.com/update/tcpdump-3.7.2-72.i586.rpm
      </location>
      <method>version number</method>
      <base>online catalogue</base>
    </update>
  </classification>
```

6.3.3 Inhalte

Für jedes Datenobjekt muss entschieden werden, ob es als atomar im Sinne der Klassifikation behandelt wird, oder aus verschiedenen, einzeln zu klassifizierenden Bestandteilen zusammengesetzt ist.

Schema

```
<xsd:complexType name="contentType">
  <xsd:sequence>
    <xsd:element name="header" minOccurs="0">
      <xsd:complexType>
```

```
        <xsd:sequence>
          <xsd:element name="classification"
            type="classificationType"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="object" type="objectType"
      maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

Mit der hier gewählten Lösung, als Bestandteile wiederum Datenobjekte (also Elemente vom im vorherigen Abschnitt beschriebenen Typ `objectType`) zu verwenden, ergibt sich ein einfacher aber leistungsfähiger Mechanismus zur Beschreibung beliebiger Inhaltsmodelle. Zusammen mit den vorhandenen Datenobjekt-Identifikationen können sowohl übliche Inhalte, wie zum Beispiel komprimierte Dateien in einer Archivdatei oder in einer Datei gespeicherte Emails, strukturiert dargestellt werden, wie auch ungewöhnlichere Fälle. Ebenso ist die Darstellung von Netzwerkpaketen in üblichen Schichtenmodellen ohne zusätzlichen Aufwand möglich.

Zusätzlich ist es möglich, für ein zusammengesetztes Datenobjekt noch einen `header` separat zu klassifizieren. Ein Header unterscheidet sich dabei von einem Datenobjekt durch die fehlende Identifikation und ist somit nicht vom enthaltenden Datenobjekt zu trennen. Diese Möglichkeit ist insbesondere für selbst-extrahierende Archivdateien notwendig, die zusätzlich zu den enthaltenen Dateien noch eine Dekompressionsroutine enthalten, die natürlich auch infiziert werden könnte. Es sind aber auch andere Anwendungen denkbar[27].

Aufgrund der hier verfolgten Logik darf der Inhalt eines Datenobjekts nicht leer sein, sondern muss wenigstens ein anderes Datenobjekt enthalten. Andernfalls wäre es möglich, „leere“ Datenobjekte ohne Klassifikation zu beschreiben. Theoretisch ist auch die Möglichkeit eines Datenobjekts, welches nur einen Header enthält, denkbar. Dies wird sinnvollerweise jedoch als ein atomares Datenobjekt behandelt, da es keine identifizierbaren Inhalte gibt.

Des Weiteren ist zu beachten, dass eine rekursive Verschachtelung von Datenobjekten nicht möglich ist. Es gibt zwar keine Einschränkungen hinsichtlich der Identifikation der enthaltenen Datenobjekte, es ist aber nicht möglich, statt einer Identifikation einen Verweis auf ein Datenobjekt einer anderen Hierarchiestufe anzugeben.

`Instanz`

[27] Etwa die Klassifikation falsch formatierter Email-Header, wenn diese einen Fehler in einem Email-Programm ausnutzen.

```
<object>
  <identification>
    <mail>...</mail>
  </identification>
  <content>

    <header>
      <classification>...</classification>
    </header>

    <object>
      <identification>
        <file>...</file>
      </identification>
      <classification>...</classification>
    </object>

  </content>
</object>
```

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Konzept für ein XML-basiertes Berichtsformat entwickelt und dieses umgesetzt. Das Berichtsformat ist über den angedachten Einsatz im aVTC bzw. zur Klassifikation von Malware hinaus auch für andere Anwendungen einsetzbar, die Software nach den hier beschriebenen Kriterien klassifizieren.

Dabei wurden die in Kapitel 4.1 genannten Anforderungen des aVTC erfüllt. Die Darstellung beliebig langer Dateinamen und -pfade ist möglich, allerdings kann dies in der Praxis durch den verwendeten XML-Parser eingeschränkt sein[1]. Durch die getrennte Darstellung von Datenobjekten und deren Inhalten (falls welche vorhanden sind), wird die Eindeutigkeit der Dateinamen nicht mehr wie bisher unnötig eingeschränkt. Dadurch lässt sich die Testauswertung weiter vereinfachen, obwohl bei der Logdatei-Konvertierung aus anderen Formaten resultierende Einschränkungen dadurch natürlich nicht behoben werden können. Zudem wird die Auswertung bisher nicht berücksichtigter qualitativer Kriterien (wie zum Beispiel der gemeldete Malware-Typ und die Malware-Plattform) möglich.

Die in Abschnitt 4.2 genannte Anforderung, Klassifikationen sowohl vorzudefinieren als auch die Verwendung nicht vordefinierter Klassifikationen zu erlauben, ist mit XML Schema nicht sinnvoll realisierbar. Deshalb wurde die Menge der verfügbaren Klassifikationen auf die in dieser Arbeit definierten Elemente `malicious`, `unknown`, `verified`, `modified` und `update` beschränkt. Allerdings lassen sich innerhalb der Klassifikation `unknown` weitere Beschreibungen der Datenobjekte frei angeben. Da diese Arbeit ursprünglich nur die Klassifikation von Malware zum Ziel hatte, ist diese Einschränkung in diesem Rahmen vertretbar. Für weiterführende Entwicklungen könnte sich aber die Notwendigkeit ergeben, das Schema erweitern zu müssen, um zusätzliche Klassifikationen zu realisieren. Die Menge der darstellbaren Zeichen ist durch die Verwendung von Unicode im XML-Standard nahezu unbeschränkt. Allerdings ist bei Implementationen darauf zu achten, dass auch eine geeignete Kodierung verwendet wird (sinnvollerweise UTF-8 oder UTF-16, da diese von jedem Parser unterstützt werden müssen).

[1] Zum Beispiel kann der Parser „Xerces“ die Datentyp-Restriktion `xsd:length` nur bis zu einem maximalen Wert von 2.147.483.647 Zeichen validieren. Größere Werte sind allerdings auch extrem unwahrscheinlich.

Die in 4.3 genannten Anforderungen konnten dagegen nicht zufriedenstellend realisiert werden. Bedingt durch die Flexibilität der Basistechnologie XML lässt sich die Einbettung maliziöser Objekte (zum Beispiel als „processing instruction“) nicht verhindern. Auch die Möglichkeit zur Darstellung beliebiger Dateinamen (oder ähnlicher Elementinhalte) bedingt die Einbettung maliziöser Inhalte, da sich keine Zeichen als ungültig ausschließen lassen. Hier liegt es also nach wie vor in der Verantwortung der interpretierenden Anwendung, die vorhandenen Inhalte als Textdaten und nicht als ausführbare Objekte zu behandeln. Gegenüber dem vorherigen Datenformat im aVTC hat sich die Lesbarkeit für Menschen (bedingt durch die „Markup“-Anteile) eher verringert. Allerdings wird dadurch eine leichtere und vor allem erstmals eindeutige Verarbeitbarkeit durch Software erreicht[2]. Auch die Forderung, die wichtigsten Daten nicht in den „Markup“-Anteilen zu „verstecken“, wurde nur bedingt umgesetzt. Die Identifikation der Datenobjekte ist zwar auch nach Verlust jeglicher Formatierung größtenteils noch verständlich. Aber die Klassifikation wird aus Effizienzgründen durch Elementnamen statt durch Elementinhalte ausgedrückt. Ebenso werden einige Zusatzangaben (wie zum Beispiel die Malware-Erkennungsmethode) aus Effizienzgründen als Attribut verwendet und würden bei Verlust jeglicher Formatierung vermutlich nicht erhalten bleiben.

Insgesamt ist das resultierende Datenformat zwar deutlich besser verarbeitbar, dies wird jedoch durch höhere Komplexität und eine etwas geringere Lesbarkeit erkauft.

Weiterführende Arbeiten könnten, abgesehen von der Verwendung in der Testauswertung des aVTC, eine Anwendung zum Ziel haben, die Berichte in diesem Format anzeigen und verwalten kann. Auf diese Weise könnte die Darstellung sicherheitsrelevanter Informationen (gerade in heterogenen Netzwerkumgebungen) deutlich vereinfacht werden.

Im folgenden Abschnitt werden noch einige Hinweise zu einem Einsatz im aVTC gegeben.

7.1 Verwendung im aVTC

Bisher wurden (wie in Abschnitt 2.3 beschrieben) aus der Logdatei eines Anti-Malware-Produkts mittels eines Perl-Skripts zwei Dateilisten extrahiert, die den Klassifikationen „infiziert“ und „nicht infiziert“ entsprechen. Die Funktion zum Schreiben der Dateilisten sieht (vereinfacht) wie folgt aus:

```
sub writeData {
  open (INFECTED, "> $outNorm");
  binmode(INFECTED);
  open (OKAY, "> $outOkay");
  binmode(OKAY);
```

[2] Von den verbesserten Interpretationsmöglichkeiten profitiert natürlich auch eine manuelle Auswertung.

```
  foreach my $sample (keys %data) {
    if (exists($data{$sample}->{"infected"})) {
      foreach my $name (keys %{$data{$sample}->{"infected"}}) {
        my @lines = @{$data{$sample}->{"infected"}->{$name}};
        foreach my $line (@lines) {
          print(INFECTED $sample, " ", $name, $EOL);
        }
      }
    }
    if (exists($data{$sample}->{"okay"})) {
      foreach my $line (@{$data{$sample}->{"okay"}}) {
        print(OKAY $sample, $EOL);
      }
    }
  }

  close (INFECTED);
  close (OKAY);
}
```

Um stattdessen ein SCL-konformes XML-Dokument zu erzeugen, könnte die Funktion auf folgende Weise abgeändert werden:

```
use XML::Writer;

sub writeData {
  my $outFile = new IO::File("> $outNorm");
  my $out = new XML::Writer(OUTPUT => $outFile,
                            DATA_MODE => "true",
                            DATA_INDENT => 2);
  $out->startTag("report",
                 "version" => "1.0",
                 "xmlsns" => "http://www.michel-messerschmidt.de/scl/v1"
                 );
  $out->startTag("origin");
  $out->dataElement("date", $date);
  $out->dataElement("program", $prod);
  $out->endTag("origin");

  foreach my $sample (keys %data) {
    my ($filename, $path) = fileparse($sample);

    if (exists($data{$sample}->{"infected"})) {
      foreach my $name (keys %{$data{$sample}->{"infected"}}) {
        my @lines = @{$data{$sample}->{"infected"}->{$name}};
        foreach my $line (@lines) {
          $out->startTag("object");
          $out->startTag("identification");
          $out->startTag("file");
```

```
          $out->dataElement("name", $filename);
          $out->dataElement("path", $path);
          $out->endTag();
          $out->endTag();
          $out->startTag("classsification");
          $out->startTag("malicious");
          $out->dataElement("mwname", $name);
          $out->endTag();
          $out->endTag();
          $out->endTag();
        }
      }
    }

    if (exists($data{$sample}->{"okay"})) {
      foreach my $line (@{$data{$sample}->{"okay"}}) {
        $out->startTag("object");
        $out->startTag("identification");
        $out->startTag("file");
        $out->dataElement("name", $filename);
        $out->dataElement("path", $path);
        $out->endTag();
        $out->endTag();
        $out->startTag("classsification");
        $out->startTag("unknown");
        $out->dataElement("property", "not infected");
        $out->endTag();
        $out->endTag();
        $out->endTag();
      }
    }
  }

  $out->endTag("report");
  $out->end();
}
```

Dabei wird vom XML::Writer Modul ein wohlgeformtes XML-Dokument erzeugt. Also werden zum Beispiel &-Zeichen in Elementinhalten automatisch durch `&` ersetzt, ohne dass dazu weitere Vorkehrungen notwendig sind. Hierbei handelt es sich natürlich nur um ein sehr vereinfachtes Beispiel, das die Vorteile des SCL-Schemas nicht wirklich ausnutzt. Für eine tatsächliche Umsetzung ist es sinnvoll, größere Teile des Perl-Skripts zu erneuern. Dadurch könnten aber insbesondere die Funktionen zur Behandlung von Archivinhalten (die zurzeit über 50% der Komplexität und Größe der Skripte ausmachen) fast komplett wegfallen. Die bisherige Erfahrung[3] zeigt, dass es nicht sinnvoll ist, solche

[3] Bisher machen die Anpassungen der produkt-spezifischen Skripte den Hauptaufwand der Testauswertung aus.

Änderungen gleich an allen produkt-spezifischen Perl-Skripten durchzuführen, da diese ohnehin bei jedem neuen Test an die unvermeidlichen Änderungen der Produkte angepasst werden müssen. Stattdessen ist es aber möglich, eine Referenz-Implementierung eines Skripts zu entwickeln, die dann im Rahmen eines neuen Tests und nach Fertigstellung des weiterentwickelten Hauptprogramms auf alle Produkte angewendet werden kann.

Zur Weiterverarbeitung solcher vereinheitlichten Logdateien im Hauptprogramm der Testauswertung existieren etliche Möglichkeiten. Zum Beispiel könnte das Instanz-Dokument mittels dem `XML::LibXML`-Modul eingelesen und direkt als Datenstruktur im Hauptprogramm bereitgestellt werden.

```
use XML::LibXML;

my $parser = XML::LibXML->new();
my $report = $parser->parse_file($logfile);
my $root = $report->getDocumentElement;
```

Mit diesen Befehlen wird die gesamte Logdatei als DOM[4]-konforme Datenstruktur in Perl bereitgestellt und kann mit den im DOM-Standard definierten Methoden verarbeitet werden (zum Beispiel kann mit der Methode „`getDocumentElement`“ auf das Wurzelelement zugegriffen werden).

Alternativ ist es mit XPath bzw. dem `XML::XPath`-Modul theoretisch auch möglich, bestimmte Daten (zum Beispiel ein „Array“ aller Dateinamen) direkt aus dem Instanz-Dokument zu extrahieren. Es existieren auch Implementationen des SAX[5]-Standards, die eine Event- bzw. Stream-basierte Verarbeitung ermöglichen. Natürlich müssen für jede dieser Möglichkeiten die zugrunde liegenden Perl-Module und (gegebenenfalls externe XML-Parser) installiert sein.

Der Aufwand ist also schon etwas größer als bei der Verwendung einfacher Textdateien, hält sich jedoch in einem vertretbaren Rahmen. Gleichzeitig lassen sich Probleme, die bisher nur sehr aufwändig zu lösen waren, wesentlich einfacher realisieren (zum Beispiel ist es jetzt nicht mehr notwendig, einen Dateinamen und -pfad daraufhin zu untersuchen, ob es sich um eine Archivdatei handelt).

Eine Implementierung sollte gleichzeitig andere Veränderungen in Angriff nehmen, die durch Verbesserungen der Testprozeduren im Laufe des letzten Jahres möglich geworden sind. Daher ist es vermutlich sinnvoller, die Kernalgorithmen auf Basis dieses XML-Datenformats komplett neu zu entwerfen. Auf diese Weise könnten einige nicht mehr benötigte Funktionen entfallen (zum Beispiel die Optimierung für alte Pentium-Rechner oder aufwändige Sortierroutinen aufgrund ungenügend vorformatierter Daten), während

[4] „Document Object Model“, ein W3C-Standard zur Verarbeitung von XML-Dokumenten (siehe `http://www.w3.org/DOM/`)

[5] “Simple API for XML“, ein weiterer Standard zur Verarbeitung von XML-Dokumenten (siehe `http://www.saxproject.org/`)

die Vorteile der direkten Verwendung der XML-Daten als Perl-interne Datenstrukturen nur auf diese Weise vollkommen genutzt werden können. Die Schnittstelle zwischen produkt-spezifischen Skripten und Hauptprogramm sollte ebenfalls überarbeitet werden, um die bisherige Einschränkung der Klassifikation auf „infizierte“ und „nicht infizierte“ Objekte zu vermeiden. Die insgesamt notwendigen Änderungen betreffen grob geschätzt ein Drittel des gesamten Programmcodes.

Deshalb lassen sich die hier vorgeschlagenen Änderungen nicht sofort implementieren, sondern erfordern erst den Entwurf eines veränderten Konzepts, das alle Umstände berücksichtigt. Ausgehend von den derzeitigen Kapazitäten des aVTC und unter Berücksichtigung des Umstands, dass alle Änderungen nebenher zu den laufenden Tests durchgeführt werden müssten, ist davon auszugehen, dass Entwurf, Implementation und Test einer aktualisierten Auswertungsumgebung einige Monate dauern werden. Zusätzlich ist auch der Installations- und Test-Aufwand für die benötigte XML-Unterstützung zu berücksichtigen.

Literaturverzeichnis

[Arbaugh 1996] Arbaugh, William; Farber, David; Smith, Jonathan: „A Secure and Reliable Bootstrap Architecture“, Proceedings of the IEEE Symposium on Security and Privacy, 1997, pp 65-71,
http://www.cis.upenn.edu/ waa/aegis.ps

[Bontchev 1998] Bontchev, Vesselin: „Methodology of Computer Anti-Virus Research“,
Dissertation, 1998, Universität Hamburg

[Brunnstein 1999] Brunnstein, Klaus: „From AntiVirus to AntiMalware Software and Beyond: Another Approach to the Protection of Customers from Dysfunctional System Behaviour“,
22. National Information Systems Security Conference, 1999,
http://csrc.nist.gov/nissc/1999/proceeding/papers/p12.pdf

[Clark 2001] Clark, James: „TREX - Tree Regular Expressions for XML“, 2001,
http://www.thaiopensource.com/trex/

[Cohen 1984] Cohen, Fred: „Computer Viruses - Theory and Experiments“, 1984,
http://www.all.net/books/virus

[FitzGerald 2003] FitzGerald, Nick: „A virus by any other name - virus naming updated“, pp. 7-9, Virus Bulletin, January 2003,
http://www.virusbtn.com/magazine/archives/200301/caro.xml

[Gordon 2003] Gordon, Sarah: „That Which We Call Rose.A“, pp. 14-17,
Virus Bulletin, March 2003

[Harold 2001] Harold, Elliotte Rusty; Means, W. Scott: „XML in a Nutshell“, 2001,
O’Reilly & Associates, ISBN 0-596-00058-8

[HTML4] W3C (World Wide Web Consortium): „HTML 4.01 Specification“,
Recommendation, 1999-12-24,
Editors: Dave Raggett, Arnaud Le Hors and Ian Jacobs,
http://www.w3.org/TR/1999/REC-html401-19991224/

[IEEE2003] IEEE (Institute of Electrical and Electronic Engineers): „Guidelines for 64-Bit global identifier (EUI-64™) registration authority“, 2003, http://standards.ieee.org/regauth/oui/tutorials/EUI64.html

[ISO 8601] ISO (International Organization for Standardization): „Representations of dates and times“, ISO/IEC 8601:1988, 1988-06-15

[ISO 19757-2] ISO (International Organization for Standardization): „Information technology – Document Schema Definition Language (DSDL) – Part 2: Regular-grammar-based validation – RELAX NG“, ISO/IEC 19757-2:2003, 2003-11-28

[Kawaguchi 2001] Kohsuke KAWAGUCHI: „W3C XML Schema Made Simple“, 2001, http://www.xml.com/pub/a/2001/06/06/schemasimple.html

[Murata 2000] MURATA Makoto: „RELAX (Regular Language description for XML)“, 2000, http://www.xml.gr.jp/relax/

[Messerschmidt 2002] Messerschmidt, Michel: „Analyse der aVTC-Tests und Verbesserung der Testauswertung“, Baccalaureatsarbeit, 2002, Universität Hamburg, http://www.michel-messerschmidt.de/mm_bacc.pdf

[Pfleeger 1997] Pfleeger, Charles P.: „Security in Computing (Second Edition)“, 1997, Prentice Hall, ISBN 0-13-185794-0

[RelaxNG] OASIS (Organization for the Advancement of Structured Information Standards): „RELAX NG Specification“, Committee Specification, 2001-12-03, Editors: James Clark and MURATA Makoto, http://www.oasis-open.org/committees/relax-ng/spec-20011203.html

[RFC 1738] IETF (Internet Engineering Task Force): Request for Comments (RFC) 1738: „ Uniform Resource Locators (URL)“, Dezember 1994, Editors: T. Berners-Lee, L. Masinter and M. McCahill, http://www.ietf.org/rfc/rfc1738.txt

[RFC 2045] IETF (Internet Engineering Task Force): Request for Comments (RFC) 2045: „Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies“, November 1996, Editors: N. Freed and N. Borenstein, http://www.ietf.org/rfc/rfc2045.txt

[RFC 2046] IETF (Internet Engineering Task Force): Request for Comments (RFC) 2046: „Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types“, November 1996,
Editors: N. Freed and N. Borenstein,
http://www.ietf.org/rfc/rfc2046.txt

[RFC 2047] IETF (Internet Engineering Task Force): Request for Comments (RFC) 2047: „Multipurpose Internet Mail Extensions (MIME) Part Three: Message Header Extensions for Non-ASCII Text“, November 1996,
Editor: K. Moore,
http://www.ietf.org/rfc/rfc2047.txt

[RFC 2048] IETF (Internet Engineering Task Force): Request for Comments (RFC) 2048: „Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures“, November 1996,
Editors: N. Freed, J. Klensin and J. Postel,
http://www.ietf.org/rfc/rfc2048.txt

[RFC 2373] IETF (Internet Engineering Task Force): Request for Comments (RFC) 2373: „IP Version 6 Addressing Architecture“, Juli 1998,
Editors: R. Hinden and S. Deering,
http://www.ietf.org/rfc/rfc2373.txt

[RFC 2396] IETF (Internet Engineering Task Force): Request for Comments (RFC) 2396: „Uniform Resource Identifiers (URI): Generic Syntax“, August 1998,
Editors: T. Berners-Lee, R. Fielding and L. Masinter,
http://www.ietf.org/rfc/rfc2396.txt

[RFC 2732] IETF (Internet Engineering Task Force): Request for Comments (RFC) 2732: „Format for Literal IPv6 Addresses in URL's“, Dezember 1999,
Editors: R. Hinden, B. Carpenter and L. Masinter,
http://www.ietf.org/rfc/rfc2732.txt

[RFC 2822] IETF (Internet Engineering Task Force): Request for Comments (RFC) 2822: „Internet Message Format“, April 2001,
Editor: P. Resnick,
http://www.ietf.org/rfc/rfc2822.txt

[Schmall 2002] Schmall, Markus: „Classification and identification of malicious code based on heuristic techniques utilizing Meta languages“,
Dissertation, 2002, Universität Hamburg

[Schneier 1996] Schneier, Bruce: „Angewandte Kryptographie“ 1996, Addison-Wesley, ISBN 3-89319-854-7

[Seedorf 2002] Seedorf, Jan: „Verfahren zur Qualitätsbestimmung der Erkennung von bösartiger Software“,
Diplomarbeit, 2002, Universität Hamburg,
http://agn-www.informatik.uni-hamburg.de/papers/doc/Diplomarbeit_JanSeedorf_pdf.zip

[SOX] W3C (World Wide Web Consortium): „Schema for Object-Oriented XML 2.0“,
Note, 1999-07-30,
Editors: A. Davidson, M. Fuchs, M. Hedin, M. Jain, J.Koistinen, C. Lloyd, M. Maloney and K. Schwarzhof,
http://www.w3c.org/TR/NOTE-SOX-19990730/

[van der Vlist 2002] van der Vlist, Eric: „XML Schema“, 2002, O’Reilly & Associates, ISBN 0-596-00252-1

[XHTML] W3C (World Wide Web Consortium): „XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)“,
Recommendation, 2000-01-26 (revised 2002-08-01),
Editors: W3C HTML Working Group,
http://www.w3.org/TR/2002/REC-xhtml1-20020801/

[XML] W3C (World Wide Web Consortium): „Extensible Markup Language (XML) 1.0 (Second Edition)“,
Recommendation, 2000-10-06,
Editors: Tim Bray, Jean Paoli, C.M. Sperberg-McQueen and Eva Maler,
http://www.w3c.org/TR/2000/REC-xml-20001006

[XML11] W3C (World Wide Web Consortium): „Extensible Markup Language (XML) 1.1“,
Candidate Recommendation, 2002-10-15,
Editor: John Cowan,
http://www.w3c.org/TR/2002/CR-xml11-20021015

[XMLNS] W3C (World Wide Web Consortium): „Namespaces in XML“,
Recommendation, 1999-01-14,
Editors: Tim Bray, Dave Hollander and Andrew Layman,
http://www.w3c.org/TR/1999/REC-xml-names-19990114

[XMLSchema-0] W3C (World Wide Web Consortium): „XML Schema Part 0: Primer“,
Recommendation, 2001-05-02,
Editor: David Fallside,
http://www.w3c.org/TR/2001/REC-xmlschema-0-20010502

[XMLSchema-1] W3C (World Wide Web Consortium): „XML Schema Part 1: Structures“,
Recommendation, 2001-05-02,
Editors: Henry Thompson, David Beech, Murray Maloney and Noah Mendelsohn,
http://www.w3c.org/TR/2001/REC-xmlschema-1-20010502

[XMLSchema-2] W3C (World Wide Web Consortium): „XML Schema Part 2: Datatypes“,
Recommendation, 2001-05-02,
Editors: Paul Biron and Ashok Malhotra,
http://www.w3c.org/TR/2001/REC-xmlschema-2-20010502

Anhang A

Das SCL Schema

Anmerkung: In einige überlange Zeilen wurden auch hier zusätzliche Zeilenumbrüche eingefügt, um die Lesbarkeit zu wahren. Dies wird mit dem Zeichen ⤸ gekennzeichnet, das somit *nicht* Teil des Schemas ist.

```
<?xml version="1.0"?>

<xsd:schema version="1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.michel-messerschmidt.de/scl/v1"
  targetNamespace="http://www.michel-messerschmidt.de/scl/v1"
  elementFormDefault="qualified" attributeFormDefault="unqualified"
  xml:lang="en">

  <xsd:annotation>
    <xsd:appinfo>
      Id: scl.xsd,v 1.75 2004/03/19 00:01:38 mic Exp
    </xsd:appinfo>
    <xsd:documentation>
      This schema is the normative source of the
      "software classification language".
      Note that comments like this one are normative parts of the
      definition and must be obeyed by conforming applications.

      The key words "MUST", "MUST NOT", "REQUIRED", "SHALL",
      "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED",  "MAY", and
      "OPTIONAL" in this document are to be interpreted as described
      in RFC 2119.
    </xsd:documentation>
  </xsd:annotation>


  <xsd:element name="report">
    <xsd:annotation>
      <xsd:documentation>
        This MUST be the root element for every conforming document.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexType>
```

```
      <xsd:sequence>
        <xsd:element name="origin" type="originType"/>
        <xsd:element name="object" type="objectType"
          maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="version" type="xsd:token" default="1.0"/>
    </xsd:complexType>
  </xsd:element>


  <xsd:complexType name="originType">
    <xsd:annotation>
      <xsd:documentation>
        The "origin" element describes the origin of a report, i.e.
        where and when it was created, etc.
        Although most of these elements are optional, it is
        recommended to use them all.
        Note that a "creator" element is missing intentionally,
        because only the creating program and system is needed to
        identify the source of a report while reporting a human user
        may violate some privacy laws.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:all>
      <xsd:element name="date" type="xsd:dateTime">
        <xsd:annotation>
          <xsd:documentation>
            This is a similiar time format as used by the XHTML
            standard. The general format is "YYYY-MM-DDThh:mm:ss"
            followed by a optional timezone specified as "Z" (for UTC),
            "+hh:mm"  or "-hh:mm" (for the difference from UTC).

            A timezone MUST be specified if it is known (to avoid
            comparison problems).
            See "XML SChema - Datatypes" 3.2.7 for an exact type
            definition and algorithms to compare dateTime values.
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="program" type="xsd:token">
        <xsd:annotation>
          <xsd:documentation>
            The program that created the report. This SHOULD be a
            constant value, so make sure to keep all changing
            attributes ("XYZ 2004") only in the "version" element.
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="system" type="xsd:token" minOccurs="0">
```

```
      <xsd:annotation>
        <xsd:documentation>
          Used to identify the system that contains the reported
          software (...that is scanned).
          It is recommended to use an IP address, hostname or
          NetBIOS name.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="version" type="xsd:token" minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>
          This should be used to give the exact version of the
          program that created a report.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="dataversion" type="xsd:token" minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>
          This SHOULD be used to identify the version of the
          data basis that is used for the classification.
          Antimalware products should insert the date/version of
          their malware definitions/patterns here.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="options" type="xsd:string" minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>
          Should be used to give all program options and settings
          that are required to reproduce a report.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="contact" type="xsd:string" minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>
          Who can be contacted (mail, phone, ...) if there are any
          questions about a report or the program that created the
          report.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
  </xsd:all>
</xsd:complexType>


<xsd:complexType name="objectType">
```

```
    <xsd:annotation>
      <xsd:documentation>
        Each "object" element identifies and classifies a data object.
        An object may contain other objects, to provides a method to
        classify separate parts of an data object.
        This SHOULD NOT be used describe each data object in its
        greatest detail but to provide only enough details to be able
        to classsify the interesting data objects.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="identification" type="identificationType"/>
      <xsd:choice>
        <xsd:annotation>
          <xsd:documentation>
            Either this is an atomic object (for the analysis) then
            only a classification is required.
            Or this object is a container for other other objects,
            that must be reported within a 'content' element.
            No overall status report is allowed in the latter case
            (Although such status reports may be contructed by the
            interpreting program, they MUST MOT be included in this
            report format).
          </xsd:documentation>
        </xsd:annotation>
        <xsd:element name="content" type="contentType"/>
        <xsd:element name="classification" type="classificationType"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>


  <xsd:complexType name="identificationType">
    <xsd:choice>
      <xsd:element name="file" type="fileIdentType"/>
      <xsd:element name="mail" type="mailIdentType"/>
      <xsd:element name="sector" type="sectorIdentType"/>
      <xsd:element name="packet" type="packetIdentType"/>
      <xsd:element name="memory"  type="memoryIdentType"/>
      <xsd:element name="octetstream" type="streamIdentType"/>
    </xsd:choice>
  </xsd:complexType>


  <xsd:complexType name="classificationType">
    <xsd:choice>
      <xsd:element name="malicious" type="malClassType"
        maxOccurs="unbounded"/>
      <xsd:element name="verified" type="goodClassType"/>
```

```
    <xsd:element name="unknown" type="unknownClassType"/>
    <xsd:element name="modified" type="modifiedClassType"/>
    <xsd:element name="update" type="updateClassType"/>
  </xsd:choice>
</xsd:complexType>


<xsd:complexType name="contentType">
  <xsd:sequence>
    <xsd:element name="header" minOccurs="0">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="classification"
            type="classificationType"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="object" type="objectType"
      maxOccurs="unbounded">
      <xsd:annotation>
        <xsd:documentation>
          Each content element MUST contain at least one object.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>



<xsd:annotation>
  <xsd:documentation>
    And now all identification types:
  </xsd:documentation>
</xsd:annotation>

<xsd:complexType name="fileIdentType">
  <xsd:sequence>
    <xsd:choice>
      <xsd:element name="uri" type="fileURI">
        <xsd:annotation>
          <xsd:documentation>
            Use this to report files by an well-formed URI
            (as defined in RFC 2396) with the "file" scheme
            (defined in RFC 1738).
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
```

```
        <xsd:sequence>
          <xsd:element name="name" type="nonEmptyString">
            <xsd:annotation>
              <xsd:documentation>
                "name" and "path" should be used to report a filename
                and its (absolute) path separatly.

                If it not possible to use separate elements, the
                complete path and filename may be reported in the name
                element (although that SHOULD be avoided). There MUST
                be no path element in this case.

                Note 1: Linebreaks are treated as part of the filename
                (like any other character too), because of the "string"
                datatype.

                Note 2: It is not required that the path ends with
                a path separator (for example "\"). For correct
                interpretation, it may be necessary to insert a path
                separator between the contents of the "path" and the
                "name" element.
              </xsd:documentation>
            </xsd:annotation>
          </xsd:element>
          <xsd:element name="path" type="nonEmptyString" minOccurs="0"/>
        </xsd:sequence>
      </xsd:choice>
      <xsd:element name="mimetype" type="mimeDatatype" minOccurs="0"/>
      <xsd:element name="creationtime" type="xsd:dateTime"
        minOccurs="0"/>
      <xsd:element name="modificationtime" type="xsd:dateTime"
        minOccurs="0"/>
      <xsd:element name="accesstime" type="xsd:dateTime"
        minOccurs="0"/>
      <xsd:element name="length" type="xsd:nonNegativeInteger"
        minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>
            Length of this file in bytes.
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="checksum" type="checksumType" minOccurs="0"
        maxOccurs="99"/>
    </xsd:sequence>
  </xsd:complexType>


  <xsd:complexType name="mailIdentType">
```

```
<xsd:sequence>
  <xsd:element name="header" minOccurs="0" maxOccurs="unbounded">
    <xsd:annotation>
      <xsd:documentation>
        Contains a mail header field as specified in RFC 2822
        (including the non-ASCII header extensions defined in
        RFC 2047).
        Header fields included here MUST be handled unfolded,
        because all whitespace characters (space = #x20,
        tab = #x09, LF = #x0A, CR = #x0D) will be replaced with
        space (#x20) by the XML parser.

        The most interesting headers for complete messages are
        "Message-ID" and "Date". For messages in MIME format it
        is recommended to include the header
        "Content-Transfer-Encoding".

        For MIME messageparts (only "Content-"... headers exist),
        it is recommended to include "Content-Transfer-Encoding".

        There is a separate element available for the
        "Content-Type" header because it is considered more
        important.

        Note that the implementation here allows headers that are
        illegal for RFC 2822 and RFC 822 because violations of
        these standards occur in current mail usage.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleType>
      <xsd:restriction base="xsd:normalizedString">
        <xsd:pattern value="[^:]+:.*"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
  <xsd:element name="mimetype" type="mimeDatatype">
    <xsd:annotation>
      <xsd:documentation>
        MIME type of a mail body or a bodypart (as defined in
        RFC 2045-2049).
        Because the "Content-Type" header is considered most
        important for classification, its value SHOULD be included
        here and not in another header element. Note that
        parameters are not allowed to be included here. If a
        parameter is important for the identification or
        classification of a mail body, it may be reported with the
        "mimeparameter" element (see below).

        If it is known that the "Content-Type" header is forged, the
```

```
        correct value SHOULD be used here. This is the only case,
        where it is useful to include an additional header element
        with the forged header.

        If no MIME type is known, the default type "text/plain"
        SHOULD be used (in accordance with RFC 2045, 5.2).
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>
  <xsd:element name="mimeparameter" minOccurs="0" maxOccurs="unbounded">
    <xsd:annotation>
      <xsd:documentation>
        This may be used to report important parameters that are
        appended to the MIME type in a "Content-Type" header.
        Commonly used parameters include "charset" for "text" types
        and "boundary" for "multipart" types.
        if there is no "charset" parameter for a "text" type, the
        default value "charset=us-ascii" may be asssumed.

        The "boundary" parameter is only useful, if the bodyparts
        of a multipart message are reported withon a "content"
        element.

        As with the header implementation, this implementation has
        much less restrictions than the RFCs due to current
        practice.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleType>
      <xsd:restriction base="xsd:normalizedString">
        <xsd:pattern value="[^=]+=.*"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
  <xsd:element name="length" type="xsd:nonNegativeInteger"
    minOccurs="0">
    <xsd:annotation>
      <xsd:documentation>
        The length of the mail body (excluding any headers)
        in bytes.
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>
  <xsd:element name="checksum" type="checksumType"
    minOccurs="0" maxOccurs="99">
    <xsd:annotation>
      <xsd:documentation>
        A checksum of the mail body (excluding any headers).
      </xsd:documentation>
```

```
      </xsd:annotation>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>


<xsd:complexType name="sectorIdentType">
  <xsd:sequence>
    <xsd:element name="device" type="xsd:token">
      <xsd:annotation>
        <xsd:documentation>
          This specifies the physical device of this sector.
          Examples: "hda", "fd0", "multi(0)disk(0)rdisk(0)"
          or "disk drive a:"
          More detailed specifications (e.g. "hda5") are allowed.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:choice>
      <xsd:element name="mbr">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="startaddress" type="sectorAddressType"
              minOccurs="0"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="single">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="startaddress"
              type="sectorAddressType"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:sequence>
        <xsd:element name="boot">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="startaddress"
                type="sectorAddressType"/>
              <xsd:element name="endaddress" type="sectorAddressType"
                minOccurs="0"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="count" type="xsd:positiveInteger">
          <xsd:annotation>
            <xsd:documentation>
```

```
              Number of sectors in this object including
              startaddress and endaddress.
            </xsd:documentation>
          </xsd:annotation>
        </xsd:element>
      </xsd:sequence>
      <xsd:sequence>
        <xsd:element name="chain" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="startaddress"
                type="sectorAddressType"/>
              <xsd:element name="endaddress"
                type="sectorAddressType"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="count" type="xsd:positiveInteger">
          <xsd:annotation>
            <xsd:documentation>
              Number of sectors in this object including
              startaddress and endaddress.
            </xsd:documentation>
          </xsd:annotation>
        </xsd:element>
      </xsd:sequence>
    </xsd:choice>
    <xsd:element name="description" type="xsd:token" minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>
          Possibility to add a description of this data object
          (for example: "partition table", "BIOS Parameter Block"
           or "Master File Table").
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="checksum" type="checksumType" minOccurs="0"
      maxOccurs="99"/>
  </xsd:sequence>
  <xsd:attribute name="size" type="xsd:positiveInteger"
    default="512">
    <xsd:annotation>
      <xsd:documentation>
        Sector length in bytes.
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
</xsd:complexType>
```

```
<xsd:complexType name="sectorAddressType">
  <xsd:sequence>
    <xsd:element name="lba" type="xsd:nonNegativeInteger">
      <xsd:annotation>
        <xsd:documentation>
          The "Logical Block Address" (LBA) of this sector.
          To be compatible with the current LBA standard,
          64 bit values must be supported.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="chs" minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>
          The old C/H/S disk sector addressing method.
          ATA-5 restricts cylinder values to 16 bit,
          head values to 4 bit and sector values to 8 bit.
          This datatype allows bigger values to make this
          addressing method useful for other disktypes too.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="diskCylinder" type="xsd:unsignedInt"/>
          <xsd:element name="diskHead" type="xsd:unsignedByte"/>
          <xsd:element name="diskSector" type="xsd:unsignedShort"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>


<xsd:complexType name="packetIdentType">
  <xsd:sequence>
    <xsd:choice maxOccurs="unbounded">
      <xsd:element name="single">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:group ref="packetStruct"/>
            <xsd:element name="checksum" type="checksumType"
              minOccurs="0" maxOccurs="99"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="sequence">
        <xsd:complexType>
          <xsd:sequence>
```

```
            <xsd:group ref="packetStruct"/>
            <xsd:element name="count" type="xsd:positiveInteger">
              <xsd:annotation>
                <xsd:documentation>
                  Number of packets in this object.
                </xsd:documentation>
              </xsd:annotation>
            </xsd:element>
            <xsd:element name="checksum" type="checksumType"
              minOccurs="0" maxOccurs="99"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>

<xsd:group name="packetStruct">
  <xsd:sequence>
    <xsd:element name="protocol" type="xsd:NMTOKEN">
      <xsd:annotation>
        <xsd:documentation>
          The name of the network protocol. May not include most
          punctuation characters (similar to XML element names).
          A common name SHOULD be used (Examples: "http", "dns",
          "tcp", "icmp", "arp").
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="source" type="packetAddressType"
      minOccurs="0"/>
    <xsd:element name="destination" type="packetAddressType"
      minOccurs="0"/>
    <xsd:element name="time" type="xsd:dateTime" minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>
          Timestamp when this packet was received or captured.
          For packet sequences, this is the timestamp of the first
          packet.
          Some application may require specification of fractions
          of a second although this is optional for the
          "xsd:dateTime" datatype.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="length" type="xsd:positiveInteger"
      minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>
```

```
          Packet length in bytes.
          For a sequence of packets this must be the sum of all
          paket lengths.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
  </xsd:sequence>
</xsd:group>

<xsd:complexType name="packetAddressType">
  <xsd:sequence>
    <xsd:element name="interface" minOccurs="0">
      <xsd:complexType>
        <xsd:choice>
          <xsd:element name="mac" type="macAddress">
            <xsd:annotation>
              <xsd:documentation>
                The MAC address MUST be given as a hexadecimal value
                (for example "123456789abc" for the MAC address
                "12:34:56:78:9a:bc").
              </xsd:documentation>
            </xsd:annotation>
          </xsd:element>
          <xsd:element name="eui64" type="eui64Address">
            <xsd:annotation>
              <xsd:documentation>
                The IETF EUI-64 address.
              </xsd:documentation>
            </xsd:annotation>
          </xsd:element>
          <xsd:element name="other" type="xsd:token">
            <xsd:annotation>
              <xsd:documentation>
                Other interface identifiers may be given with this
                element.
              </xsd:documentation>
            </xsd:annotation>
          </xsd:element>
        </xsd:choice>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="host">
      <xsd:complexType>
        <xsd:choice>
          <xsd:element name="ip" type="ipAddress">
            <xsd:annotation>
              <xsd:documentation>
                The IP address MUST be given in hexadecimal
                ("c0a82a00") or dotted-decimal format
```

```
                ("192.168.42.0").
              </xsd:documentation>
            </xsd:annotation>
          </xsd:element>
          <xsd:element name="ipv6" type="ipv6Address">
            <xsd:annotation>
              <xsd:documentation>
                The IPv6 address MUST be given as hexadecimal value
                or in a format defined in RFC 2372 (see declaration
                of datatype "ipv6adddress for examples).
              </xsd:documentation>
            </xsd:annotation>
          </xsd:element>
          <xsd:element name="dnsname" type="xsd:NMTOKEN">
            <xsd:annotation>
              <xsd:documentation>
                The DNS hostname
              </xsd:documentation>
            </xsd:annotation>
          </xsd:element>
          <xsd:element name="winsname" type="xsd:token">
            <xsd:annotation>
              <xsd:documentation>
                The WINS / NetBIOS hostname.
              </xsd:documentation>
            </xsd:annotation>
          </xsd:element>
          <xsd:element name="other" type="xsd:token">
            <xsd:annotation>
              <xsd:documentation>
                Any other host identification method.
              </xsd:documentation>
            </xsd:annotation>
          </xsd:element>
        </xsd:choice>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="service" type="xsd:NMTOKEN" minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>
          The service address SHOULD be specified by a common
          service name ("http") or as a decimal port number
          (for example "80" for http).
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

```
<xsd:complexType name="memoryIdentType">
  <xsd:sequence>
    <xsd:element name="startaddress" type="xsd:token">
    </xsd:element>
    <xsd:element name="endaddress" type="xsd:token" minOccurs="0"/>
    <xsd:element name="length" type="xsd:positiveInteger">
      <xsd:annotation>
        <xsd:documentation>
          Length of this memory range in bytes. This includes the
          startaddress and the endaddress.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="description" type="xsd:token" minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>
          Possibility to add a description of this data object.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="checksum" type="checksumType" minOccurs="0"
      maxOccurs="99"/>
  </xsd:sequence>
</xsd:complexType>


<xsd:complexType name="streamIdentType">
  <xsd:sequence>
    <xsd:element name="source" type="xsd:token" minOccurs="0"/>
    <xsd:element name="destination" type="xsd:token" minOccurs="0"/>
    <xsd:element name="length" type="xsd:positiveInteger">
      <xsd:annotation>
        <xsd:documentation>
          Length of this octectstream in bytes.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="mimetype" type="mimeDatatype" minOccurs="0"/>
    <xsd:element name="description" type="xsd:token" minOccurs="0">
      <xsd:annotation>
        <xsd:documentation>
          Possibility to add a description of this data object.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="checksum" type="checksumType" maxOccurs="99"/>
  </xsd:sequence>
</xsd:complexType>
```

```
<xsd:annotation>
  <xsd:documentation>
    And now all classification types:
  </xsd:documentation>
</xsd:annotation>

<xsd:complexType name="malClassType">
  <xsd:annotation>
    <xsd:documentation>
      There are different possibilities to give a classification
      for a malicious object:
        - use an ID from a database like CVE or Bugtraq
        - use a CARO name
        - use different elements to specify the various name parts:
          type, platform, (familiy) name, variant and (all other)
          modifiers. It is possible to use only some of these parts.
        - use only the "malwarename" element with arbitrary content.
          This is NOT RECOMMENDED because it hides the internal name
          structure and requires more external logic to interpret
          such reports correctly.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:choice>
      <xsd:element name="id">
        <xsd:annotation>
          <xsd:documentation>
            May be used to specify an ID of a malware or vulnerability
            database (for example a CVE or Bugtraq ID).
            The attribute "type" MUST identify the database.
          </xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
          <xsd:simpleContent>
            <xsd:extension base="xsd:token">
              <xsd:attribute name="type" type="xsd:token"
                use="required"/>
            </xsd:extension>
          </xsd:simpleContent>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="caroname" type="caroNameType"/>
      <xsd:sequence>
        <xsd:element name="mwtype" type="noSpaceToken" minOccurs="0">
          <xsd:annotation>
            <xsd:documentation>
```

```
        One of the following strings SHOULD be used
        (if appropriate):
          virus
          worm
          intended
          trojan
          generator
          dropper
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>
  <xsd:element name="mwplatform" type="noSpaceToken"
    minOccurs="0">
    <xsd:annotation>
      <xsd:documentation>
        One of the common names or abbreviations for
        malware platforms SHOULD be used (as listed in the
        revised CARO naming scheme).
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>
  <xsd:element name="mwname" type="xsd:token">
    <xsd:annotation>
      <xsd:documentation>
        If used together with all other name part elements, this
        SHOULD contain only the malware family.
        "unknown" MUST be used as name for any malware that
        can't be identified by a more exact name.

        Although whitespace is allowed in malware names for
        compatibility reasons, it SHOULD NOT be used without
        very good reason.
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>
  <xsd:element name="mwvariant" type="noSpaceToken"
    minOccurs="0">
    <xsd:annotation>
      <xsd:documentation>
        Variants are commonly enumerated by using uppercase
        letters: A, B, ..., Z, AA, AB, .., ZZ, AAA, ...
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>
  <xsd:element name="mwmodifier" type="noSpaceToken"
    minOccurs="0" maxOccurs="unbounded">
    <xsd:annotation>
      <xsd:documentation>
        This should be used for any other name parts
```

```
              like @mm suffixes, language codes, etc.
            </xsd:documentation>
          </xsd:annotation>
        </xsd:element>
      </xsd:sequence>
    </xsd:choice>
    <xsd:element name="property" type="xsd:token" minOccurs="0"
      maxOccurs="unbounded">
      <xsd:annotation>
        <xsd:documentation>
          This is a possibility to report other object properties,
          that are considered important.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="method" type="mwMethodType"
    default="exact identification">
      <xsd:annotation>
        <xsd:documentation>
          The method used to detect or identify this malware.
        </xsd:documentation>
      </xsd:annotation>
  </xsd:attribute>
</xsd:complexType>


<xsd:simpleType name="mwMethodType">
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="exact identification"/>
    <xsd:enumeration value="generic identification"/>
    <xsd:enumeration value="heuristic detection"/>
    <xsd:enumeration value="other"/>
  </xsd:restriction>
</xsd:simpleType>


<xsd:simpleType name="caroNameType">
  <xsd:annotation>
    <xsd:documentation>
      This datatype is based on the revised CARO naming scheme
      (as published by Nick FitzGerald in Virus Bulletin Jan 2003).
      The complete structure must match the syntax:
        TYPE://PLATFORM/FAMILY.GROUP.VARIANT[MODIFIERS]
      Every name part may contain only the characters: A-Za-z0-9_-
      Allowed modifiers are (in this order):
      :LOCALE
      #PACKER
      @M or @MM
```

```
        !VENDOR_SPECIFIC_COMMENT

        The following regular expressions are used for the name parts:
        TYPE, PLATFORM, FAMILY, GROUP, PACKER: [-_A-Za-z0-9]{1,20}
        LOCALE: ([A-Za-z]{2}|[Uu][Nn][Ii])
        VARIANT: (Inflen|SubvarDevo?)
        Inflen: [0-9]{1,20}
        Subvar: [A-Za-z]{1,20}
        Devo: [0-9]{1,20}
        @M[M]: @[Mm][Mm]?
        VENDOR_SPECIFIC_COMMENT: .*

        Multiple TYPE or PLATFORM parts may be given as comma-separated
        list that is enclosed in braces (defined by the additional
        patterns).
      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="noSpaceToken">
      <xsd:pattern value="[-_A-Za-z0-9]{1,20}://[-_A-Za-z0-9]{1,20}/
[-_A-Za-z0-9]{1,20}(\.[-_A-Za-z0-9]{1,20})?\.([0-9]{1,20}|[A-Za-z]{1
,20}[0-9]{0,20})((:([A-Za-z]{2}|[Uu][Nn][Ii]))?(#[-_A-Za-z0-9]{1,20}
)?(@[Mm][Mm]?)?(!.*)?)?"/>
      <xsd:pattern value="[-_A-Za-z0-9]{1,20}://\{[-_A-Za-z0-9]{1,20
},[-_A-Za-z0-9]{1,20})+\}/[-_A-Za-z0-9]{1,20}(\.[-_A-Za-z0-9]{1,20}
)?\.([0-9]{1,20}|[A-Za-z]{1,20}[0-9]{0,20})((:([A-Za-z]{2}|[Uu][Nn][
Ii]))?(#[-_A-Za-z0-9]{1,20})?(@[Mm][Mm]?)?(!.*)?)?"/>
      <xsd:pattern value="\{[-_A-Za-z0-9]{1,20}(,[-_A-Za-z0-9]{1,20}
)+\}://[-_A-Za-z0-9]{1,20}/[-_A-Za-z0-9]{1,20}(\.[-_A-Za-z0-9]{1,20}
)?\.([0-9]{1,20}|[A-Za-z]{1,20}[0-9]{0,20})((:([A-Za-z]{2}|[Uu][Nn][
Ii]))?(#[-_A-Za-z0-9]{1,20})?(@[Mm][Mm]?)?(!.*)?)?"/>
      <xsd:pattern value="\{[-_A-Za-z0-9]{1,20}(,[-_A-Za-z0-9]{1,20}
)+\}://\{[-_A-Za-z0-9]{1,20}(,[-_A-Za-z0-9]{1,20})+\}/[-_A-Za-z0-9]{
1,20}(\.[-_A-Za-z0-9]{1,20})?\.([0-9]{1,20}|[A-Za-z]{1,20}[0-9]{0,20
})((:([A-Za-z]{2}|[Uu][Nn][Ii]))?(#[-_A-Za-z0-9]{1,20})?(@[Mm][Mm]?)
?(!.*)?)?"/>
    </xsd:restriction>
  </xsd:simpleType>


  <xsd:complexType name="goodClassType">
    <xsd:annotation>
      <xsd:documentation>
        WARNING: Because this is something not in use today,
        the structure and datatype contained here MAY change
        in future versions
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="type" type="xsd:token" minOccurs="0">
```

```
        <xsd:annotation>
          <xsd:documentation>
            This optional element may be used together with the
            element "name" to give hints about the purpose/usage of
            this software object on the computer system.
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="name" type="xsd:token" minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>
            This optional element may be used together with the
            element "type" to give hints about the purpose/usage of
            this software object on the computer system.
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="method" type="xsd:token">
        <xsd:annotation>
          <xsd:documentation>
            The verification method. For example "certificate" or
            "checksum"
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="trustbase" type="xsd:token">
        <xsd:annotation>
          <xsd:documentation>
            Identifies the data basis, that is used to verify the
            software object.
            For example the issuer of a certificate, if you're
            verifing software by cryptographic signatures.
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>


  <xsd:complexType name="unknownClassType">
    <xsd:sequence>
      <xsd:element name="property" minOccurs="0" maxOccurs="unbounded">
        <xsd:annotation>
          <xsd:documentation>
            This is a possibility to specify the presence or absence of
            known security relevant object properties.
            Example properties:
              "not infected"
              "not analysed"
```

```
            "may be a corrupted file"
            "contains inactive malicious code fragments"
            "signature verification failed"

          NOTE: "not analysed" has the special meaning, that the
                data object has not been analysed. This property
                MUST be used, whenever there was a problem analysing
                a data object.

                "not infected" SHOULD be used if it is desired to
                report objects that are found to be "clean" (which
                means that the analysis is sufficient to be sure that
                the object is not infected by some replicating
                malware). This is similar to the "ok" messages used
                by many antimalware products.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
        <xsd:simpleContent>
          <xsd:extension base="xsd:token">
            <xsd:attribute name="reason" use="optional">
              <xsd:annotation>
                <xsd:documentation>
                  This SHOULD always be used if an object can not be
                  analysed. Examples:
                    "compression format not supported"
                    "object is encrypted"
                    "time limit reached"
                    "size limit reached"
                </xsd:documentation>
              </xsd:annotation>
            </xsd:attribute>
          </xsd:extension>
        </xsd:simpleContent>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>


<xsd:complexType name="modifiedClassType">
  <xsd:sequence>
    <xsd:sequence maxOccurs="unbounded">
      <xsd:element name="found" type="xsd:token" >
        <xsd:annotation>
          <xsd:documentation>
            Report a modified object property here.
          </xsd:documentation>
        </xsd:annotation>
```

```
        </xsd:element>
        <xsd:element name="expected" type="xsd:token">
          <xsd:annotation>
            <xsd:documentation>
              Report the expected value here.
            </xsd:documentation>
          </xsd:annotation>
        </xsd:element>
      </xsd:sequence>
      <xsd:element name="method" type="xsd:token">
        <xsd:annotation>
          <xsd:documentation>
            The verification method. For example: "checksum"
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="base" type="xsd:token">
        <xsd:annotation>
          <xsd:documentation>
            Identifies the base of the modification checks against
            which the objects where checked. For example:
            "local database" or "online catalogue"
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>


  <xsd:complexType name="updateClassType">
    <xsd:sequence>
      <xsd:element name="current" type="xsd:token">
        <xsd:annotation>
          <xsd:documentation>
            Identify the curently installed version.
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="new" type="xsd:token">
        <xsd:annotation>
          <xsd:documentation>
            Identify the up-to-date version.
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="location" type="xsd:anyURI" minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>
            Location where the update is available.
```

```
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="method" type="xsd:token">
      <xsd:annotation>
        <xsd:documentation>
          The method used to determine pending updates.
          For example: "checksum" or "version number"
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element name="base" type="xsd:token">
      <xsd:annotation>
        <xsd:documentation>
          Identifies the source of the list of up-to-date objects.
          For example: "online catalogue"
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>



<xsd:annotation>
  <xsd:documentation>
    Here are customized datatypes for attribute values and
    simple element content.
  </xsd:documentation>
</xsd:annotation>


<xsd:simpleType name="nonEmptyString">
  <xsd:annotation>
    <xsd:documentation>
      Defines a string that must contain at least one character.
      This is used to exclude only the emtpy string from the
      value space.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string">
    <xsd:minLength value="1"/>
  </xsd:restriction>
</xsd:simpleType>


<xsd:simpleType name="noSpaceToken">
  <xsd:annotation>
```

```
      <xsd:documentation>
        Defines a string without any whitespace characters ("space",
        "tab","line feed" and "carriage return").
      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:pattern value="\S*"/>
    </xsd:restriction>
  </xsd:simpleType>


  <xsd:simpleType name="hexNumber">
    <xsd:annotation>
      <xsd:documentation>
        Defines a hexadecimal number. Uppercase and lowercase letters
        are allowed as well as leading zeros.
        Note that this definition includes decimal numbers.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:pattern value="[0-9a-fA-F]+"/>
    </xsd:restriction>
  </xsd:simpleType>


  <xsd:simpleType name="macAddress">
    <xsd:annotation>
      <xsd:documentation>
        Defines a 48-bit MAC interface address either as a single
        hexadecimal value or as hexadecimal byte values separated by
        colon or hyphen (for example "12:34:56:78:9A:BC").
      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:pattern value="[0-9a-fA-F]{12}"/>
      <xsd:pattern value="[0-9a-fA-F]{2}([-:][0-9a-fA-F]{2}){5}"/>
    </xsd:restriction>
  </xsd:simpleType>


  <xsd:simpleType name="eui64Address">
    <xsd:annotation>
      <xsd:documentation>
        Defines a 64-bit EUI-64 interface address either as a single
        hexadecimal value or as hexadecimal byte values separated by
        colon or hyphen (for example "01-23-45-67-89-AB-CD-EF").
        (See http://standards.ieee.org/regauth/oui/tutorials/EUI64.html
        for details about EUI-64).
      </xsd:documentation>
```

```
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:pattern value="[0-9a-fA-F]{16}"/>
      <xsd:pattern value="[0-9a-fA-F]{2}([-:][0-9a-fA-F]{2}){7}"/>
    </xsd:restriction>
  </xsd:simpleType>


  <xsd:simpleType name="ipAddress">
    <xsd:annotation>
      <xsd:documentation>
        Defines an IP address of a single host in either hexadecimal
        (for example "c0a80101") or dotted-decimal notation
        (for example "192.168.1.1").
      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:pattern value="[0-9a-fA-F]{8}"/>
      <xsd:pattern value="(0?\d?\d|[1]\d\d|2[0-5][0-5])(\.(0?\d?\d|
[1]\d\d|2[0-5][0-5])){3}"/>
    </xsd:restriction>
  </xsd:simpleType>


  <xsd:simpleType name="ipv6Address">
    <xsd:annotation>
      <xsd:documentation>
        Defines an IPv6 address of a single host either as 128-bit
        hexadecimal number or as one of the notations defined in
        RFC 2373.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:pattern value="[0-9a-fA-F]{32}">
        <xsd:annotation><xsd:documentation>
            Matches a hexadecimal value. For example
            "1080000000000000000088000200C417A"
          </xsd:documentation></xsd:annotation>
      </xsd:pattern>
      <xsd:pattern value="[0-9a-fA-F]{1,4}(:[0-9a-fA-F]{1,4}){7}">
        <xsd:annotation><xsd:documentation>
            Matches the syntax defined in RFC 2373, section 2.2.1
            For example "1080:0:0:0:8:800:200C:417A"
          </xsd:documentation></xsd:annotation>
      </xsd:pattern>
      <xsd:pattern value="([0-9a-fA-F]{1,4}(:[0-9a-fA-F]{1,4}){0,6})
?::([0-9a-fA-F]{1,4}(:[0-9a-fA-F]{1,4}){0,6})?">
        <xsd:annotation><xsd:documentation>
            Matches an address in compressed form as defined in
```

```
            RFC 2373, section 2.2.2.
            For example "1080::8:800:200C:417A"
          </xsd:documentation></xsd:annotation>
      </xsd:pattern>
      <xsd:pattern value="[0-9a-fA-F]{1,4}(:[0-9a-fA-F]{1,4}){5}:(0?↵
\d?\d|[1]\d\d|2[0-5][0-5])(\.(0?\d?\d|[1]\d\d|2[0-5][0-5])){3}">
        <xsd:annotation><xsd:documentation>
            Matches a mixed IPv6/IPv4 notation as defined in
            RFC 2373, section 2.2.3.
            For example "1080:0:0:0:8:800:32.12.65.122"
          </xsd:documentation></xsd:annotation>
      </xsd:pattern>
      <xsd:pattern value="([0-9a-fA-F]{1,4}(:[0-9a-fA-F]{1,4}){0,4})↵
?::([0-9a-fA-F]{1,4}(:[0-9a-fA-F]{1,4}){0,4}):(0?\d?\d|[1]\d\d|2[0-5↵
][0-5])(\.(0?\d?\d|[1]\d\d|2[0-5][0-5])){3}">
        <xsd:annotation><xsd:documentation>
            Matches allowed combinations of sections 2.2.2 and 2.2.3
            For example "1080::8:800:32.12.65.122"
          </xsd:documentation></xsd:annotation>
      </xsd:pattern>
      <xsd:pattern value="([0-9a-fA-F]{1,4}(:[0-9a-fA-F]{1,4}){0,4})↵
?::(0?\d?\d|[1]\d\d|2[0-5][0-5])(\.(0?\d?\d|[1]\d\d|2[0-5][0-5])){3}">
        <xsd:annotation><xsd:documentation>
            Matches allowed combinations of sections 2.2.2 and 2.2.3
            For example "1080::32.12.65.122"
          </xsd:documentation></xsd:annotation>
      </xsd:pattern>
    </xsd:restriction>
  </xsd:simpleType>


  <xsd:simpleType name="fileURI">
    <xsd:annotation>
      <xsd:documentation>
        A URI scheme for file locations as defined in RFC 1738.
        Because xsd:anyURI already defines a general URI syntax
        it is sufficient to restrict only the scheme prefix here.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:anyURI">
      <xsd:pattern value="file://.*"/>
    </xsd:restriction>
  </xsd:simpleType>


  <xsd:complexType name="checksumType">
    <xsd:annotation>
      <xsd:documentation>
        This datatype holds a checksum for a specified algorithm
```

```
      (given as attribut value). The algorithm must be one of the
      following list.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:simpleContent>
    <xsd:extension base="hexNumber">
      <xsd:attribute name="type" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:token">
            <xsd:enumeration value="SHA1"/>
            <xsd:enumeration value="MD5"/>
            <xsd:enumeration value="RIPEMD"/>
            <xsd:enumeration value="HAVAL"/>
            <xsd:enumeration value="SNEFRU"/>
            <xsd:enumeration value="GOST"/>
            <xsd:enumeration value="MD4"/>
            <xsd:enumeration value="MD2"/>
            <xsd:enumeration value="CRC32"/>
            <xsd:enumeration value="CRC16"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>


<xsd:simpleType name="mimeDatatype">
  <xsd:annotation>
    <xsd:documentation>
      The syntax of MIME content types is defined in RFC 2045,
      section 5.1. As specified every token consists of one or more
      ASCII characters with the exclusion of control characters,
      SPACE, the characters ()@,;:\"/[]?= and &lt; and &gt;

      To avoid problems with unicode characters this restriction is
      expressed as an list of explicitly allowed characters here:
      0-9A-Z!#$%'*+-.^_`a-z{|}~ and &amp;

      The top level media types are restricted to the types defined
      in RFC 2045, section 5.1 and RFC 2046 including the "X-"
      extension method): text, image, audio, video, application,
      message, multipart
      ATTENTION: Top level media types defined according to the IETF
      registration process (described in RFC 2048) are NOT included
      here and can't be described with this schema version.

      RFC 2045 allows media types to be appended with parameters
      (separated by semicolon, for example:
```

```
        "text/plain; charset=us-ascii")
        Parameters are not allowed here to keep things simple.
        Instead there is an additional element for mime parameters,
        that may be used if parameters are considered important for
        the current task (see below for the "mimeParameterType").

        The regular expression is more complex, because RFC 2045
        allows all uppercase/lowercase combinations for media types.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
      <xsd:pattern value="[tT][eE][xX][tT]/[-0-9A-Za-z!#%'_`~\$\*\+\.\^\{\|\}&amp;]+"/>
      <xsd:pattern value="[iI][mM][aA][gG][eE]/[-0-9A-Za-z!#%'_`~\$\*\+\.\^\{\|\}&amp;]+"/>
      <xsd:pattern value="[aA][uU][dD][iI][oO]/[-0-9A-Za-z!#%'_`~\$\*\+\.\^\{\|\}&amp;]+"/>
      <xsd:pattern value="[vV][iI][dD][eE][oO]/[-0-9A-Za-z!#%'_`~\$\*\+\.\^\{\|\}&amp;]+"/>
      <xsd:pattern value="[aA][pP][pP][lL][iI][cC][aA][tT][iI][oO][nN]/[-0-9A-Za-z!#%'_`~\$\*\+\.\^\{\|\}&amp;]+"/>
      <xsd:pattern value="[mM][eE][sS][sS][aA][gG][eE]/[-0-9A-Za-z!#%'_`~\$\*\+\.\^\{\|\}&amp;]+"/>
      <xsd:pattern value="[mM][uU][lL][tT][iI][pP][aA][rR][tT]/[-0-9A-Za-z!#%'_`~\$\*\+\.\^\{\|\}&amp;]+"/>
      <xsd:pattern value="[Xx]-[-0-9A-Za-z!#%'_`~\$\*\+\.\^\{\|\}&amp;]+/[-0-9A-Za-z!#%'_`~\$\*\+\.\^\{\|\}&amp;]+"/>
    </xsd:restriction>
  </xsd:simpleType>


</xsd:schema>
```

Anhang B

Ein gültiges Beispieldokument

```
<?xml version="1.0" encoding="UTF-8"?>

<report version="1.0" xmlns="http://www.michel-messerschmidt.de/scl/v1">

  <origin>
    <date>2004-12-01T23:09:00+01:00</date>
    <system>192.168.42.8 (Suse Linux 9.0)</system>
    <program>handmade</program>
    <version>
      Id: example.xml,v 1.38 2004/03/19 00:02:00 mic Exp
    </version>
    <dataversion>none</dataversion>
  </origin>


  <object>
    <identification>
      <file>
        <name>sample.zip</name>
        <path>/local/malware/</path>
        <mimetype>application/zip</mimetype>
        <checksum type="MD5">60c99a8e5cf56620b581244c4bd21714</checksum>
      </file>
    </identification>
    <content>

      <header>
        <classification>
          <unknown>
            <property>not infected</property>
          </unknown>
        </classification>
      </header>

      <object>
```

```
    <identification>
      <file>
        <name>sample.exe</name>
        <path>w32/Klez/H/</path>
        <mimetype>application/x-msdos-program</mimetype>
        <checksum type="MD5">
          f6a69fe04b358f65ee5e126473169801
        </checksum>
      </file>
    </identification>
    <classification>
      <malicious method="exact identification">
        <!-- The "method" attribute may be omitted here, because-->
        <!-- "exact identification" is the default value.       -->
        <mwtype>virus</mwtype>
        <mwplatform>W32</mwplatform>
        <mwname>Klez</mwname>
        <mwvariant>H</mwvariant>
      </malicious>
    </classification>
  </object>

  <object>
    <identification>
      <file>
        <name>sample2.exe</name>
        <path>w32/Klez/H/</path>
        <mimetype>application/x-msdos-program</mimetype>
        <checksum type="MD5">
          337d1c41ff87a878197a8d10b76b9539
        </checksum>
      </file>
    </identification>
    <classification>
      <unknown>
        <property>may be a corrupted file</property>
        <property>
          contains inactive malicious code fragments
        </property>
      </unknown>
    </classification>
  </object>

  <object>
    <identification>
      <file>
        <!-- a path specification may be missing -->
        <name>sample3.doc</name>
        <mimetype>application/msword</mimetype>
```

```
          <creationtime>2003-12-23T14:40:03+01:00</creationtime>
          <modificationtime>
            2004-01-16T11:08:34+01:00
          </modificationtime>
          <accesstime>2004-01-16T11:08:34+01:00</accesstime>
        </file>
      </identification>
      <classification>
        <malicious>
          <caroname>virus://W97M/Melissa.A@mm</caroname>
        </malicious>
      </classification>
    </object>

    <object>
      <identification>
        <file>
          <name>sample4.doc</name>
          <path>o97m/doc</path>
          <mimetype>application/msword</mimetype>
        </file>
      </identification>
      <classification>
        <malicious>
          <caroname>virus://{W97M,X97M}/Tristate.C</caroname>
        </malicious>
      </classification>
    </object>

    <object>
      <identification>
        <file>
          <name>sample5.exe</name>
          <path>newvir/</path>
          <mimetype>application/x-msdos-program</mimetype>
        </file>
      </identification>
      <classification>
        <malicious>
          <caroname>{trojan,dropper}://W32/Backdoor.XYZ</caroname>
        </malicious>
      </classification>
    </object>

  </content>
</object>


<object>
```

```
    <identification>
      <file>
        <uri>file://localhost/C:/malware/sample.rar</uri>
        <mimetype>application/x-rar-compressed</mimetype>
        <length>340179</length>
        <checksum type="SHA1">
          2f90bb1f9be68eecd28b715bfa2e9310de13812e
        </checksum>
      </file>
    </identification>
    <classification>
      <unknown>
        <property reason="compression format not supported">
          not analysed
        </property>
      </unknown>
    </classification>
  </object>


  <object>
    <identification>
      <file>
        <name>cmd.exe</name>
        <path>C:\windows</path>
        <mimetype>application/x-msdos-program</mimetype>
        <checksum type="SHA1">
          13be37723c374b95f18a059dd9ae1aee2119e98a
        </checksum>
      </file>
    </identification>
    <classification>
      <verified>
        <type>system core application</type>
        <name>Command interpreter</name>
        <method>signature</method>
        <trustbase>Microsoft root certificate</trustbase>
      </verified>
    </classification>
  </object>


  <object>
    <identification>
      <mail>
        <header>From: "Infected User" user1@example.com</header>
        <header>To: user2@example.org</header>
        <header>
          Subject: I send you this file in order to have your advice
```

```
      </header>
      <header>date: Sat, 20 Oct 2001 15:42:11 +0200</header>
      <header>
        Message-ID: &lt;01234567.89ABCDEF@example.com&gt;
      </header>
      <header>MIME-Version: 1.0</header>
      <mimetype>multipart/mixed</mimetype>
      <mimeparameter>boundary="1234567890"</mimeparameter>
    </mail>
  </identification>
  <content>
    <!-- There is no requirements to report all message parts.    -->
    <!-- This is done here just for completeness of this example. -->
    <object>
      <identification>
        <mail>
          <header>Content-Transfer-Encoding: quoted-printable</header>
          <mimetype>text/plain</mimetype>
          <mimeparameter>charset=ISO-8859-1</mimeparameter>
          <length>91</length>
          <checksum type="MD5">
            6f695fca1f1ef736c7f594d432de4371
          </checksum>
        </mail>
      </identification>
      <classification>
        <unknown>
          <property>not infected</property>
          <property>possibly created by mass-mailing worm</property>
        </unknown>
      </classification>
    </object>
    <object>
      <identification>
        <mail>
          <header>Content-Transfer-Encoding: base64</header>
          <mimetype>application/mixed</mimetype>
          <mimeparameter>name=Demo.doc.bat</mimeparameter>
          <length>157184</length>
          <checksum type="MD5">
            37a69526f514f7d9fa97f88914276f83
          </checksum>
        </mail>
      </identification>
      <classification>
        <malicious>
          <mwtype>worm</mwtype>
          <mwplatform>W32</mwplatform>
          <mwname>Sircam</mwname>
```

```
          <mwmodifier>@MM</mwmodifier>
        </malicious>
      </classification>
    </object>
  </content>
</object>


<!-- It is perfectly legal to classify a complete   -->
<!-- mail message even if it has multiple bodyparts -->
<object>
  <identification>
    <mail>
      <header>From: "Infected User" user1@example.com</header>
      <header>To: user3@example.org</header>
      <header>Subject: Re:</header>
      <header>date: Sat, 20 Oct 2001 16:42:11 +0200</header>
      <header>
        Message-ID: &lt;89ABCDEF.01234567@example.com&gt;
      </header>
      <header>MIME-Version: 1.0</header>
      <mimetype>multipart/related</mimetype>
      <length>39791</length>
      <checksum type="MD5">87e8a2b0db14174e6eb295f3814985b2</checksum>
    </mail>
  </identification>
  <classification>
    <malicious>
      <mwtype>worm</mwtype>
      <mwplatform>W32</mwplatform>
      <mwname>Badtrans</mwname>
      <mwvariant>B</mwvariant>
      <mwmodifier>@MM</mwmodifier>
    </malicious>
  </classification>
</object>


<object>
  <identification>
    <sector>
      <device>hda</device>
      <mbr/>
      <checksum type="MD5">32c2841701931942130ff21aa9ab20c1</checksum>
    </sector>
  </identification>
  <classification>
    <malicious>
      <!-- It is possible to use only some of the name-part elements. -->
```

```
      <!-- But this makes the interpretation more complex, because    -->
      <!-- part separators like "/" and "." must be recognized.       -->
      <mwtype>virus</mwtype>
      <mwname>Boot/Parity.B</mwname>
    </malicious>
  </classification>
</object>


<object>
  <identification>
    <sector>
      <device>fd0</device>
      <mbr>
        <startaddress><lba>0</lba></startaddress>
      </mbr>
      <checksum type="MD5">b0d9bc3fea1bd97b9388ded4bbe93ca8</checksum>
    </sector>
  </identification>
  <classification>
    <unknown>
      <property>not infected</property>
    </unknown>
  </classification>
</object>


<object>
  <identification>
    <sector size="2352">
      <device>scsi0,0</device>
      <boot>
        <startaddress><lba>64</lba></startaddress>
        <endaddress><lba>66</lba></endaddress>
      </boot>
      <count>3</count>
      <description>DOS boot code</description>
    </sector>
  </identification>
  <classification>
    <malicious method="heuristic detection">
      <mwtype>virus</mwtype>
      <mwname>unknown</mwname>
    </malicious>
  </classification>
</object>


<object>
```

```
    <identification>
      <sector>
        <device>multi(0)disk(0)rdisk(0)partition(1)</device>
        <chain>
          <startaddress><lba>70</lba></startaddress>
          <endaddress><lba>112</lba></endaddress>
        </chain>
        <count>43</count>
        <description>File Allocation Table</description>
      </sector>
    </identification>
    <classification>
      <malicious>
        <mwtype>virus</mwtype>
        <mwname>Dir_II</mwname>
      </malicious>
    </classification>
  </object>


  <object>
    <identification>
      <sector>
        <device>multi(0)disk(0)rdisk(0)partition(1)</device>
        <single>
          <startaddress><lba>1048510</lba></startaddress>
        </single>
      </sector>
    </identification>
    <classification>
      <malicious>
        <mwtype>virus</mwtype>
        <mwname>Dir_II</mwname>
      </malicious>
    </classification>
  </object>


  <object>
    <identification>
      <packet>
        <sequence>
          <protocol>tcp</protocol>
          <source>
            <interface><mac>12:34:56:78:9a:bc</mac></interface>
            <host><ip>192.168.1.1</ip></host>
            <service>32776</service>
          </source>
          <destination>
```

```
          <interface><eui64>12-34-56-ff-ff-cb-a9-87</eui64></interface>
          <host><ip>192.168.1.2</ip></host>
          <service>631</service>
        </destination>
        <length>918</length>
        <count>10</count>
        <checksum type="MD5">
          6f2a464628809e9d896793892d589c9d
        </checksum>
      </sequence>
    </packet>
  </identification>
  <classification>
    <malicious>
      <id type="CVE">CAN-2003-0195</id>
      <property>exploit: cups - denial of service</property>
    </malicious>
  </classification>
</object>


<object>
  <identification>
    <packet>
      <single>
        <protocol>tcp</protocol>
        <source>
          <host><ip>192.168.100.1</ip></host>
          <service>32776</service>
        </source>
        <destination>
          <host><ipv6>::192.168.1.1</ipv6></host>
          <service>135</service>
        </destination>
      </single>
      <single>
        <protocol>tcp</protocol>
        <source>
          <host><ip>192.168.100.2</ip></host>
          <service>32777</service>
        </source>
        <destination>
          <host><ipv6>0:0:0:0:0:0:192.168.1.1</ipv6></host>
          <service>137</service>
        </destination>
      </single>
      <single>
        <protocol>tcp</protocol>
        <source>
```

```
      <host><ip>192.168.100.3</ip></host>
      <service>32778</service>
    </source>
    <destination>
      <host><ipv6>FEC0:0:0:1:0:0:0:1</ipv6></host>
      <service>139</service>
    </destination>
  </single>
  <single>
    <protocol>tcp</protocol>
    <source>
      <host><ip>192.168.100.1</ip></host>
      <service>32779</service>
    </source>
    <destination>
      <host><ipv6>fec0:0:0:1::1</ipv6></host>
      <service>445</service>
    </destination>
  </single>
  <single>
    <protocol>tcp</protocol>
    <source>
      <host><ip>192.168.100.2</ip></host>
      <service>32780</service>
    </source>
    <destination>
      <host><ipv6>fec0:0000:0000:0001:0000:0000:0000:0001</ipv6></host>
      <service>111</service>
    </destination>
  </single>
  <single>
    <protocol>tcp</protocol>
    <source>
      <host><ip>192.168.100.3</ip></host>
      <service>32781</service>
    </source>
    <destination>
      <host><ipv6>fec00000000000010000000000000001</ipv6></host>
      <service>2049</service>
    </destination>
  </single>
  <single>
    <protocol>tcp</protocol>
    <source>
      <host><ip>192.168.100.1</ip></host>
      <service>32782</service>
    </source>
    <destination>
      <host><ip>c0a80101</ip></host>
```

```
          <service>80</service>
        </destination>
      </single>
      <single>
        <protocol>tcp</protocol>
        <source>
          <host><ip>192.168.100.2</ip></host>
          <service>32782</service>
        </source>
        <destination>
          <host><ip>C0A80101</ip></host>
          <service>25</service>
        </destination>
      </single>
      <single>
        <protocol>tcp</protocol>
        <source>
          <host><ip>192.168.100.3</ip></host>
          <service>32783</service>
        </source>
        <destination>
          <host><ip>192.168.1.1</ip></host>
          <service>22</service>
        </destination>
      </single>
    </packet>
  </identification>
  <classification>
    <unknown>
      <property>possibly portscan</property>
    </unknown>
  </classification>
</object>


<object>
  <identification>
    <memory>
      <startaddress>13005000</startaddress>
      <endaddress>130050ff</endaddress>
      <length>256</length>
      <checksum type="MD5">084e6294e33aabae550fa4718922dbd1</checksum>
    </memory>
  </identification>
  <classification>
    <malicious method="generic identification">
      <mwtype>worm</mwtype>
      <mwplatform>W32</mwplatform>
      <mwname>SQLSlammer</mwname>
```

```
    </malicious>
  </classification>
</object>


<object>
  <identification>
    <octetstream>
      <length>123456</length>
      <checksum type="MD5">a7050cc80c0a8c92da8346a3edc7e0cf</checksum>
    </octetstream>
  </identification>
  <classification>
    <malicious method="heuristic detection">
      <mwname>unknown</mwname>
    </malicious>
  </classification>
</object>


<object>
  <identification>
    <file>
      <name>tcpdump</name>
      <path>/usr/sbin</path>
      <mimetype>application/x-executable</mimetype>
      <checksum type="MD5">16b579e3a0bbbe776dcac711afeb7471</checksum>
    </file>
  </identification>
  <classification>
    <update>
      <current>3.7.2-36</current>
      <new>3.7.2-72</new>
      <location>
        ftp://vendor.example.com/update/tcpdump-3.7.2-72.i586.rpm
      </location>
      <method>version number</method>
      <base>online catalogue</base>
    </update>
  </classification>
</object>


<object>
  <identification>
    <file>
      <name>ps</name>
      <path>/bin/</path>
      <mimetype>application/x-executable</mimetype>
```

```
        <checksum type="MD5">1f62a7af07af53d5fed8d6a7f0b5879b</checksum>
      </file>
    </identification>
    <classification>
      <modified>
        <found>mtime: 2004-01-19 15:56</found>
        <expected>mtime: 2003-09-23 19:03</expected>
        <found>md5: 1f62a7af07af53d5fed8d6a7f0b5879b</found>
        <expected>md5: 852d7ca0f51415a4ee39bf90eaadb49a</expected>
        <method>checksum</method>
        <base>local database: /media/cdrom/tw.db</base>
      </modified>
    </classification>
  </object>

</report>
```

Erklärung:

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.