Technical Report

21. Oktober 2013

Sulley vs Peach:

Teil 1: Datenmodelle

Florian Schmidt, B.Sc.

Prof. Dr. Rohrmair

Hochschule Augsburg

Zusammenfassung

Es existieren verschiedene Fuzzing-Programme und -Frameworks, deren Syntax und Semantik voneinander abweichen. Spezifikationen von Protokollen und anderen Schnittstellen sind deshalb inkompatibel und müssen für jede Implementierung separat erstellt werden. Das Ziel des vorliegenden Berichts ist es zu beschreiben wie das Datenmodell einer vorliegenden Spezifikation automatisiert in das Format eines anderen Fuzzing-Frameworks transformiert werden kann. Hierzu wurde eine Anwendung erstellt, die Datenmodelle von zwei Fuzzing-Frameworks, Sulley und Peach, ineinander konvertiert. Eine automatische Konvertierung der Datenmodelle ist teilweise möglich. Eine korrekte Transformation gewisser Strukturen kann aber nur durch Benutzerinteraktion oder Erweiterungen des Ziel-Frameworks erreicht werden.

Inhaltsverzeichnis

1 Einleitung

Fuzzing oder Fuzz-Testing ist eine Black-Box Testmethode bei der Anwendungsschnittstellen und Protokollstacks durch den Input semi-valider Daten auf Schwachstellen hin untersucht werden. Es wurde zuerst 1988 von Barton Miller in einer Projektbeschreibung an seine Studenten erwähnt [20, im Vorwort]. Seit Ende der neunziger Jahre des letzten Jahrhunderts erfährt es eine immer größere Verbreitung und wird sowohl in Open-Source Projekten, als auch in der Industrie zur Detektion von Softwarefehlern genutzt.

1.1 Ziel des Berichts

In vorliegendem Bericht werden die Datenmodellierungstechniken der beiden Fuzzing-Frameworks *Peach* und *Sulley* miteinander verglichen. Ziel des Berichts ist zu zeigen, dass die Datenmodelle der beiden APIs ineinander transformiert werden können. Zu diesem Zweck wurde der Prototyp einer Anwendung zur Konvertierung der Modelle implementiert. Weitere Features, wie Zustandsmodellierung und die Beschreibung des Testablaufs, sind nicht Gegenstand dieses Berichts und werden im zweiten Teil beschrieben und erläutert.

1.2 Begrifflichkeiten

Für die Verständlichkeit des Berichts ist es notwendig einige Begriffe zu erläutern.

1.3 Fuzzing Frameworks

Als Fuzzing-Framework wird die Kombination einer API (Advanced Programming Interface) zur Generierung von Daten- und Zustandsmodellen und einer Ausführungsumgebung bezeichnet. Sie bieten zum einen die Möglichkeit Testdaten und -abläufe in eigener Syntax und Semantik zu modellieren. Zum anderen ermöglichen sie es, durch Werkzeuge wie Prozess- und Netzwerkmonitore, Testläufe zu überwachen und zu analysieren.

1.3.1 Datenmodell (oder Data Model)

Ein Datenmodell ist das abstrakte Modell einer Datenstruktur. Abhängig vom Framework erfolgt die Definition eines Datenmodells entweder im Programmcode oder deklarativ (XML).

1.3.2 Datenelement

Datenelemente sind die Bestandteile eines Datenmodells und sind eine abstrakte Darstellung von primitiven und komplexen Datentypen. Es existieren auch Datenelemente, sogenannte Blöcke, die selbst wieder Datenelemente enthalten können. Dies erlaubt die Modellierung hierarchischer Datenstrukturen.

1.3.3 Data Rendering

Das Rendern eines Datenmodells ist die Generierung von Daten aus dem abstrakten Modell während eines Fuzz-Tests.

1.3.4 Mutation

Die Werte der Datenelemente eines Datenmodells können während eines Testlaufs automatisiert verändert werden. Diese Änderungen werden jeweils vor einem Rendering-Schritt durchgeführt, sodass die Wertänderungen des Datenmodells auch in den generierten Daten widergespiegelt werden.

1.3.5 Data Cracking

Als *Data Cracking* wird das Einlesen von (Binär-)Daten und die Umwandlung von diesen Daten zu einem Datenmodell bezeichnet. Dies kann während eines Testlaufs notwendig sein, wenn die Antwort eines getesteten System interpretiert werden muss, wie zum Beispiel bei Authentifizierungen. Somit stellt das *Data Cracking* das Gegenteil des *Data Rendering* dar.

Sulley und Peach

Sulley [3] und Peach [10] sind zwei verbreitete Vertreter von Fuzzing-Frameworks. Sulley benutzt Datenstrukturen und Syntax, die denen der Fuzzing-API von SPIKE sehr ähnlich sind [19]. Peach hingegen ist eine eigenständige Entwicklung, mit Fokus auf Modularisierung.

Features

Beide Frameworks sind block-basiert, das heißt dass die Datenmodelle als eine erweiterte Form eines Top-Level-Blocks implementiert sind. Verschiedene Konfigurationsoptionen erlauben es die Generierung der Daten vor oder während eines Testlaufs zu modifizieren. Peach und Sulley bieten ebenfalls die Möglichkeit, Zustandsmodelle zu definieren. Zustandsmodelle, oder State Models, beschreiben die Kombinationen von Datenmodellen, die zur Durchführung eines Fuzz-Tests benötigt werden. Auch verfügen beide Frameworks über die Möglichkeit Testläufe mit sogenannten Monitoren, Programme die Testläufe und deren Ergebnisse aufzeichnen, zu überwachen. Peach erlaubt sowohl das Fuzzing von Netzwerkprotokollen als auch von Anwendungsschnittstellen. Sulley ist im Gegensatz dazu hauptsächlich für das Testen von Netzwerkprotokollen bestimmt, kann aber mit Zusatzsoftware so modifiziert werden, dass auch Anwendungs-Fuzzing möglich ist (siehe 2).

Tabelle 1: Features

Fuzzer	Type	Data Model	State Model	Monitoring
Sulley	Network	✓	✓	✓
Peach	General	✓	✓	✓

2 Sulley

Sulley ist ein Python-basiertes Fuzzing-Framework, dass von der 2005 von Pedram Amini ins Leben gerufenen *Open Reverse Code Engineering Community (OpenRCE)* herausgegeben wird. Das Framework lehnt sich in seiner Syntax und Semantik zu einem großen Teil an das in C implementierte Fuzzer-Construction-Kit *SPIKE* (verfügbar unter [1]) an. Die Spezifikation eines Fuzz-Tests erfolgt mittels eines Python-Skripts, indem das Daten- und das Zustandsmodell definiert wird. Die grundsätzliche Ausrichtung von Sulley liegt auf dem Fuzzing von Netzwerkprotokollen, allerdings ist auch Anwendungs- beziehungsweise Filefuzzing unter der Zuhilfenahme von Werkzeugen (PAIMEIfilefuzz) aus dem *OpenRCE / PaiMei Reverse Engineering Framework* [2] möglich.

2.1 Datenmodell

Ein Datenmodell in Sulley enthält Datenelemente verschiedenen Typs. Diese Datenelemente lassen sich in zwei Kategorien einteilen: einerseits Datenelemente mit primitivem Datentyp, wie *String*, *Number* et cetera, andererseits dem komplexen Datentyp *Block*. Ein Block kann selbst sowohl verschiedene Datenelemente primitiven Typs, als auch wieder Blöcke enthalten. Dies ermöglicht eine hierarchische Strukturierung des Datenmodells und wird als block-basierte Protokollrepräsentation bezeichnet.

2.1.1 Initialisierung

Zur Definition eines Datenmodells wird der Befehl *s_initialize* aufgerufen, einziger Parameter ist der Name des Datenmodells.

```
s_initialize('DataModel1')
```

Codeauflistung 1: Initialisierung

Nachfolgend initialisierte Datenelemente werden dem Datenmodell zugeordnet, bis durch erneuten Aufruf von *s_initialize* ein neues Datenmodell

initialisiert wird. Diesem werden dann wiederum die danach initialisierten Elemente zugeordnet.

```
s_initialize('DataModel1')
# Initialisiert ein Element vom Typ 'string' mit dem Wert 'String1' und ordnet das Element
    dem Datenmodell 'DataModel1' zu.
s_string('String1')
s_string('String2')

s_initialize('DataModel2')
s_string('String3')
s_string('String4')

s_initialize('DataModel3')
s_string('String5')
s_string('String6')
```

Codeauflistung 2: Zuordnung von Elementen

2.1.2 Datenelemente

Sulley bietet eine Reihe von verschiedenen Datenelementen, die zur Erstellung eines Datenmodells verwendet werden können. Die Elementtypen umfassen sowohl primitive Typen, wie Zahlen oder Zeichenketten, als auch komplexere Datentypen, wie Blöcke. Folgende Elementtypen stehen zur Verfügung.

- string
- delim
- byte
- word
- sword
- dword
- qword
- static
- binary

- random
- legos
- block (+ group / checksum / size / dep / encoder / repeat)

Im nachfolgenden Abschnitt 2.2 werden die verschiedenen Datenelemente im Detail beschrieben.

2.2 Datenelemente

Datenelemente sind die Bestandteile eines Datenmodells. Wie bereits in 2.1.2 erwähnt können Datenmodelle aus Datenelementen primitiven oder komplexeren Typs bestehen. Primitive Datentypen dienen dazu Werte eines Protokolls oder einer Schnittstelle abzubilden deren Typ den bekannten Typen höherer Programmiersprachen entspricht. Komplexe Typen, wie Blöcke, erlauben hingegen Datenstrukturen wie Arrays, Xml-Elemente und ähnliches abzubilden. Alle Datenelemente, egal ob primitiv oder komplex, besitzen ein Feld *name* dem ein eindeutiger Name zugewiesen werden kann, um das Datenelement identifizieren und referenzieren zu können. Zusätzlich besitzen alle Datenelemente das Feld *value*, welches den Wert des Elementes enthält. Zusätzlich enthält jedes Datenelement das Feld *fuzzable*, dass angibt ob der Wert des Datenelements während des Testlaufs mutiert werden kann. Der Standardwert ist grundsätzlich *True*, hängt aber vom Elementtyp ab. Bis auf die Elemente *group*, *random_ data* und *static*, besitzen alle Datenelemente das Array *fuzz_ library*. Es enthält Variationen des Werts des jeweiligen Datenelements (auch bekannt als Heuristiken). Die Werte des *fuzz_ library*-Arrays werden während eines Tests zyklisch durchlaufen und anstelle des eigentlichen Wertes des Datenelementes gesetzt. Auf diese Weise werden bestimmte Wertebereiche während eines Testlaufs automatisch abgearbeitet.

2.2.1 Zeichenketten (Strings)

String ist der am häufigsten auftauchende Elementtyp, da die meisten Protokolle viele Felder diesen Typs enthalten. Ein Datenelement vom Typ *string*

wird mit Hilfe der Funktion *s_string* initialisiert. Alle String-Elemente besitzen ein Feld *size* das die Länge des Strings angibt, wird der Standardwert -1 verwendet, ist die Größe des Strings dynamisch. Besitzt der String eine fixe Größe, kann zusätzlich ein Pad-Character (*padding*) angegeben werden, mit dem der String auf die vorgegebene Größe aufgefüllt wird. Das Standardzeichen für das Padding ist *\x00*. Des Weiteren kann die Kodierung (*encoding*) eines Strings angegeben werden, dies ist wichtig um die korrekte Darstellung und Verarbeitung des Strings innerhalb des Testkontextes zu gewährleisten. Die Standardeinstellung für die Kodierung ist ASCII (*ascii*), es sind aber alle Python-Standardkodierungen [14] möglich.

```
s_string('fuzz', encoding="utf-8")
```

Codeauflistung 3: String

2.2.1.1 Trennzeichen (Delimiter) Neben normalen Strings, die beliebige Zeichen enthalten können, stellt Sulley ein weiteres Datenelement, *delim*, zur Verfügung. Elemente dieses Typs werden durch Aufruf der Funktion *s_delim* initialisiert und sind für Strings-Elemente vorgesehen die ausschließlich Trennzeichen enthalten. Die Benutzung von *delim* hat den Vorteil, dass es spezielle Mutationen des Werts, wie Wiederholung, Ersetzung und Ausschluss, bereitstellt und somit das Fuzzing von Trennzeichen automatisiert.

```
s_delim(' ')
```

Codeauflistung 4: Delimiter

2.2.2 Zahlen und Binärdaten

Um binäre und numerische Werte abzubilden stellt Sulley eigene Elementtypen bereit.

2.2.2.1 bit_field Der Typ *bit_field* dient zur Modellierung binärer Daten beliebiger Länge. Zur Initialisierung dient die entsprechende Funktion, *s_bit_field*. Zur Definition der Länge der Binärdaten dient der Parameter

width. Die Byte-Reihenfolge der Binärdaten kann über den Parameter *endian* angegeben werden. „<“ steht hierbei für Little Endian und „>“ steht für Big Endian, die Default-Byte-Reihenfolge ist Little Endian. Die Eigenschaft *format* erlaubt es darüber hinaus zu definieren, ob die Binärdaten während des Testlaufs als numerische Werte (*format=ascii*) oder als Hexadezimalwerte (*format=binary*) kodiert werden. Mit Hilfe des Parameters *signed* kann spezifiziert werden, ob ein *bit_field* vorzeichenbehaftet ist. Per Default ist dieser Wert gleich *False*. Schließlich kann mit dem Parameter *full_range* angegeben werden, ob die Binärdaten während des Testlaufs über den gesamten Wertebereich mutiert werden soll. Der Standardwert für diesen Parameter ist *False* und sollte nur in Ausnahmefällen auf *True* gesetzt werden, da der Wertebereich von beispielsweise acht Byte Binärdaten bereits circa 18 Trillionen mögliche Werte umfasst. Falls nicht der gesamte Wertebereich getestet werden soll generiert Sulley nur bestimmte Werte (Maximalwert, Hälfte des Maximalwerts und so weiter) während des Tests.

```
s_bit_field(0x000cce333225, 48)
```

Codeauflistung 5: BitField

2.2.2.2 byte, word, dword, qword Um bekannte Zahlentypen darstellen zu können existieren in Sulley mehrere Elementtypen, die alle von *bit_field* abgeleitet sind. Im Gegensatz zur Basisklasse besitzen die Zahlentypen jeweils eine fixe Länge. Die Typen sind *byte*, für Zahlen mit Länge von einem Byte, *word* für Zahlen von zwei Byte Länge, *dword* mit vier Byte Länge und schließlich *qword* mit acht Byte Länge. Jeder dieser Typen kann wieder durch eine entsprechende Funktion (*s_byte*, *s_word*, *s_dword*, *s_qword*) initialisiert werden. Im Gegensatz zum Typ *bit_field* ist das Standardformat der vier Zahlentypen *ascii*. Der Unterschied zwischen den beiden Kodierungsarten kann anhand der Formatierung der Zahl 100 verdeutlicht werden: wird die ASCII-Formatierung genutzt bleibt die Darstellung des Wertes unverändert, wird die binäre Formatierung angewendet ist der resultierende Wert *x64*. Alle weiteren Parameter entsprechen denen des zugrundeliegenden Typen *bit_field*.

```
s_byte(0, format="ascii", full_range=True)
s_word(12, endian=">" format="binary")
s_dword(0xDEADBEEF, format="ascii", signed=True)
s_qword(0, format="ascii", signed=False)
```

Codeauflistung 6: Zahlen

2.2.3 Statische Datenelemente

Sulley stellt weiter Datenelemente für spezielle Zwecke zur Verfügung. Hierzu zählt das Element *static*, welches mit der Funktion *s_static* initialisiert werden kann. Es erlaubt nicht mutierbare (*fuzzable=False*), also während des Testlaufs unveränderliche, Werte beliebiger Länge zu deklarieren. Der Datentyp *static* kann auch zur Darstellung unveränderbarer, binärer Daten im Hexadezimalformat genutzt werden. Um ein *static*-Element mit Binärdaten zu füllen, kann die Initialisierungsfunktion emphs_binary genutzt werden, die eine automatische Formatierung der Binärdaten in ein von Sulley interpretierbares Format durchführt. Angaben zur Formatierung oder der Byte-Reihenfolge, wie es bei *bit_field* möglich ist (siehe 2.2.2.1), sind allerdings nicht möglich.

```
s_static('Do not fuzz me.')
s_binary('0xde 0xad be ef \xca fe 00 01 02 0xba0xdd f0 0d')
```

Codeauflistung 7: Statische Elemente

2.2.4 Zufallswerte

Zur Generierung von Zufallswerten stellt Sulley den Elementtyp *random_data* zur Verfügung, welches über die Funktion *s_random* initialisiert wird. Die minimale und maximale Länge des Zufallswertes muss über die Parameter *min_length* und *max_length* angegeben werden. Setzt man die minimale Länge gleich der maximalen Länge erhält man einen Zufallswert statischer Länge. Zusätzlich kann über den Parameter *num_mutations* angegeben werden, wie viele Mutationen auf den Startwert angewendet werden sollen. Standardmäßig wird der Wert 25-mal mutiert.

```
s_initialize('TheDataModel')
s_random('1111', min_length=4, max_length=6, num_mutations=50)
```

Codeauflistung 8: Random

2.2.5 Gruppen

Um das Testen ähnlicher Datenstrukturen zu erleichtern, besitzt Sulley den Elementtyp *group*. Die Initialisierung eines solchen Elements erfolgt durch den Aufruf von *s_ group*. Diese Initialisierungsfunktion besitzt den Parameter *name*, der im Gegensatz zu anderen Elementtypen ein Pflichtfeld ist. Dem zweiten Parameter *values*, muss eine Liste von String-Werten zugewiesen werden. Während der Ausführung eines Fuzz-Test wird über diese Liste iteriert, das heißt bei jedem Aufruf des Datenmodells wird der nächste String aus der Werteliste in die resultierende Datenstruktur eingefügt. Wird beim Testlauf der letzte Wert der Liste erreicht, beginnt die Iteration beim nächsten Aufruf des Datenmodells wieder beim ersten Element.

```
s_initialize('TheDataModel')
s_group('Group1' values=['Value1', 'Value2', 'Value3']
s_delim('_')
s_string('StringValue')
```

Codeauflistung 9: Gruppe

Der mehrmalige Aufruf des obigen Datenmodells während eines Testlaufs würde in folgenden Datenstrukturen resultieren:

```
# Erster Aufruf
'Value1_StringValue'

# Zweiter Aufruf
'Value2_StringValue'

# Dritter Aufruf
'Value3_StringValue'
```

Codeauflistung 10: Resultierende Daten

2.2.6 Blöcke

Primitive Datenelemente können zusammengefasst werden, hierzu dient das Element *block*. Um einen neuen Block zu initialisieren wird die Funktion *s_block_start* aufgerufen. Der erste Parameter beim Aufruf muss der gewünschte Name des Blocks sein. Zusätzlich sind weitere optionale Parameter vorhanden, mit denen die Generierung der Daten angepasst werden kann.

2.2.6.1 Kodierung (encoder) Die Option *encoder* erlaubt es eine Funktion anzugeben, die die Daten des Blocks in eine gewünschte Kodierung, wie zum Beispiel *Base64*, überführt. Die referenzierte Funktion darf nur einen Parameter, zur Übergabe der zu kodierenden Daten, besitzen.

```
#stub of an encoder-function
def encoderFunction(renderedBlockContent):
       ...
       return encodedBlockContent

#referencing the encoder-function
s_block_start('block1', encoder=encoderFunction)
...
s_block_end()
```

Codeauflistung 11: Kodierung

2.2.6.2 Abhängigkeiten (dep) Sulley erlaubt die dynamische Generierung von Blöcken in Abhängigkeit eines anderen primitiven Datenelements. Hierzu existieren die Optionen *dep*, *dep_value* beziehungsweise *dep_values* und *dep_compare*. Mit *dep* wird angegeben von welchem Datenelement die Generierung des Blocks abhängig ist. Mit *dep_value* kann der Wert angegeben werden, den das referenzierte Datenelement annehmen muss, die Semantik der Option *dep_values* ist analog, hier können allerdings mehrere Werte angegeben werden. *dep_compare* erlaubt es schließlich den Vergleichsoperator für die Bedingung zu definieren. Standard ist hier ==, aber alle Python-Vergleichsoperatoren können genutzt werden. Dies ermöglicht eine bedingte Generierung von Blöcken, das heißt Blöcke können je nach Wert des referenzierten Datenelements unterschiedliche Datenelemente enthalten.

```
s_short('opcode', full_range=True)

# Opcode 10 erwartet eine Authenfizierungssequenz
if(s_block_start('auth', dep='opcode', dep_value=10):
        s_string('USER')
        s_delim(' ')
        s_string('pedram')
        s_static('\r\n')
        s_string('PASS')
        s_delim(' ')
        s_delim('fuzzywuzzy')
s_block_end()

# Opcode 15 und 16 erwarten den Hostnamen als Einzelstring
if(s_block_start('hostname', dep='opcode', dep_values=[15, 16]):
        s_string('pedram.openrce.org')
s_block_end()

# Die restlichen Opcodes erwarten zwei Unterstriche gefolgt von einem String
if(s_block_start('something', dep='opcode', dep_values=[10, 15, 16], dep_compare='!='):
        s_string('__')
        s_string('some string')
s_block_end()
```

Codeauflistung 12: Abhängigkeiten [16]

In der Codeauflistung 12 wird ein Element vom Typ *short* mit dem Namen *opcode* deklariert, dessen Wert als Bedingung für die Generierung der Blöcke dient (ersichtlich durch die Referenzierung von *opcode* mittels des Parameters *dep*). Während des Testlaufs wird abhängig vom Wert des *opcode*-Elements einer der drei definierten Blöcke ausgewählt. Der Block *auth* wird gewählt, wenn der Wert des *opcode*-Elements 10 beträgt. Ist der Wert des *opcode*-Elements hingegen 15 oder 16 (dep_values=[15, 16]) wird der Block *hostname* selektiert. Für alle anderen Werte von *opcode* wird schließlich der Block *something* gewählt.

Gruppe (group) Der Parameter *group* erlaubt es einen Block mit einer Gruppe 2.2.5 zu verknüpfen. Während der Ausführung eines Fuzz-Tests hat dies zur Konsequenz, das bei jeder Auswahl eines anderen Strings aus der Werteliste der Gruppe der verknüpfte Block auf deinen Ausgangswert zurückgesetzt und neu mutiert wird. Somit wird für jedes Gruppenelement

der gesamte Wertebereich des Blocks durchschritten, was zu einer höheren Code-Abdeckung des Tests führt.

2.2.6.3 Hilfsobjekte für Blöcke Zusätzlich zu den Parametern von *s_block_start* liefert Sulley eine Reihe an Hilfsobjekten für Blöcke mit.

Sizers (size) Mittels des Objektes *size* lässt sich die Größe eines Blocks in das Datenmodell integrieren. Dieses Element wird meist verwendet, wenn der referenzierte Block in eine komplexere Datenstruktur eingebunden werden soll und dazu die Berechnung eines Offsets notwendig ist. Die Initialisierung eines *size*-Objekts erfolgt wiederum über eine korrespondierende Funktion namens *s_size*. Der Parameter *block_name* dient zur Referenzierung des Blocks, dessen Größe berechnet werden soll. Mittels *math* kann die Funktion für die Längenberechnung angegeben werden. Der Boolean-Parameter *inclusive* definiert, ob die Länge (Parameter *length*), des *sizers* bei der Längenberechnung berücksichtigt werden soll. Weiterhin können *sizers*, analog zu Nummer-Datentypen, siehe 2.2.2, durch die Parameter *endian*, *signed* und *format* konfiguriert werden.

```
s_initialize("DataModel")
s_size("block1", length=4, endian=">")
if s_block_start("block1"):
    s_string("someString")
    s_byte("\x00")
s_block_end()
```

Codeauflistung 13: size

Prüfsummen / Hashes (checksum) Zur Berechnung der Prüfsumme oder des Hashwerts eines Blocks, beinhaltet Sulley die Funktion *s_checksum*. Über den Parameter *block_name* kann der Block angegeben werden, dessen Prüfsumme oder Hash berechnet werden soll. Der Parameter *algorithm* definiert den zu verwendenden Algorithmus zur Berechnung der Prüfsumme. Verfügbare Algorithmen sind *CRC32*, *Adler32*, *MD5* und *SHA1*. Mittels *length* kann die Länge der Prüfsumme definiert werden. Der Default-Wert ist

0 und impliziert die automatische Berechnung der Länge der Prüfsumme beziehungsweise des Hashes. Die Byte-Reihenfolge kann analog zu Zahlentypen über den Parameter *endian* definiert werden, der Standard-Wert ist Little Endian (oder „<“ siehe 2.2.2).

```
s_initialize("DataModel")
s_checksum("block1", algorithm="sha1", endian=">")
if s_block_start("block1"):
    s_string("someString")
    s_byte("\x00")
s_block_end()
```

Codeauflistung 14: checksum

Wiederholung (repeat) Ein *repeat*-Element dient dazu den Block bei Ausführung des Fuzz-Test zu vervielfachen, um so zum Beispiel Pufferüberläufe zu provozieren. Die Initialisierungsfunktion *s_repeat* verfügt über die Parameter *min_reps* und *max_reps* über die die Anzahl der Wiederholungen des Blocks definiert werden kann. Der Parameter *step* gibt an mit welcher Schrittgröße das Intervall, das durch *min_reps* und *max_reps* definiert ist, durchlaufen wird. Der Name des Blocks der wiederholt werden soll, muss wie bei den anderen Block-Hilfsobjekten über dem Parameter *block_name* angegeben werden.

```
s_initialize("DataModel")
if s_block_start("block1"):
    s_delim(">", name="delim", fuzzable=False)
    s_string("test", name="string", fuzzable=False)
    s_byte(0xde, name="byte", fuzzable=False)
    s_word(0xdead, name="word", fuzzable=False)
    s_dword(0xdeadbeef, name="dword", fuzzable=False)
    s_qword(0xdeadbeefdeadbeef, name="qword", fuzzable=False)
    s_random(0, 5, 10, 100, name="random", fuzzable=False)
s_block_end()
s_repeat("block1", min_reps=5, max_reps=15, step=5)
```

Codeauflistung 15: repeat

In der Codeauflistung 15 wird das *repeat*-Element mit dem Block *block1* verknüpft. Beim ersten Aufruf des Datenmodells wird der Block fünfmal wiederholt, da die Eigenschaft *min_reps* auf 5 gesetzt wurde. Beim nächsten

Aufruf wird der Block zehnmal wiederholt, da die Schrittweite über die Eigenschaft *step* ebenfalls auf 5 gesetzt wurde. Beim nächsten Aufruf wird der Block somit fünfzehn-mal wiederholt, da wiederum die Anzahl der Wiederholungen um die Schrittweite erhöht wird. Hier ist die maximale Anzahl von Wiederholungen, definiert durch *max_ reps*, erreicht. Somit wird beim nächsten Aufruf wieder mit der Anzahl an Wiederholungen definiert in *min_ reps* fortgefahren.

2.2.7 Eigene Datenelemente (legos)

Der block-basierte Ansatz von Sulley vereinfacht dass Erstellen von benutzerdefinierten Datentypen. Durch Hinzufügen von Datelementen zu einem Standard-Sulley-Block kann nach dem Baukastenprinzip (daher der Begriff *legos*) ein neuer, komplexer Datentyp definiert werden. Folgendes einfaches Beispiel demonstriert die Erstellung eines *tag*-Datentyps zur Modellierung von Xml-Tags.

```
class tag (blocks.block):
    def __init__ (self, name, request, value, options={}):
        blocks.block.__init__(self, name, request, None, None, None, None)

        self.value = value
        self.options = options

        if not self.value:
            raise sex.error("MISSING LEGO.tag DEFAULT VALUE")

        # <example>
        # [delim][string][delim]

        self.push(primitives.delim("<"))
        self.push(primitives.string(self.value))
        self.push(primitives.delim(">"))
```

Codeauflistung 16: tag

Im der obigen Codeauflistung 16 wird eine neue Klasse *tag* deklariert, die ein Tag in einem XML-Dokument modelliert. *tag* ist abgeleitet vom Typ *block* und kann somit Datenelemente enthalten. Als Datenelemente werden die öffnenden und schließenden Klammern des Tags als *delim*-Objekte und

der Tag-Name als *string* hinzugefügt, außerdem wird auch noch die *value*-Eigenschaft des *tag*-Objektes mit dem Tag-Namen belegt.

3 Peach

Peach, herausgegeben von DejaVu Security, ist ein stark modularisiertes Fuzzing-Framework. Peach wurde ursprünglich, wie Sulley, in Python entwickelt. Für die aktuelle Version wurde eine Neuimplementierung in C# durchgeführt. Gegenstand der Untersuchung ist Peach mit der Versionsnummer 3.0.207. Die bessere Performance von (JIT-)kompiliertem .NET/Mono-Code im Vergleich zu interpretiertem Python-Skript erlaubt es die Dauer der Testläufe signifikant zu verringern. Zur Definition der Spezifikation wird, sowohl für die aktuelle als auch ältere Versionen nahezu identisches XML-Markup benutzt. Die Dateien die die XML-Definition eines Protokolls enthalten, werden von den Peach-Entwicklern gemeinhin als Peach-Pits bezeichnet.

3.1 Datenmodell

Datenmodelle in Peach sind ebenso block-basiert, wie Datenmodelle in Sulley. Jedes Datenmodell besitzt einen Namen, sowie zusätzliche Attribute. Zu diesen Zählen das Attribut *ref* dass es erlaubt andere Datenmodelle zu referenzieren, ...

```
<DataModel name="ParentModel">
  <String value="Hello " />
</DataModel>

<DataModel name="HelloWorldModel" ref="ParentModel" >
  <String value=" world!" />
</DataModel>
```

Codeauflistung 17: Referenzierung

... dabei erben sie die Datenelemente des referenzierten Modells.

```
<DataModel name="HelloWorldModel">
  <String value="Hello " />
  <String value=" world!" />
</DataModel>
```

Codeauflistung 18: Resultierende interne Darstellung

Ein weiteres Attribut ist *constraint* das einen Python-Ausdruck enthalten kann, der eine Bedingung beschreibt, die nach dem Parsen des Datenmodells gelten muss. Das Attribut *mutable* gibt an, ob das Datenmodell während des Testlaufs verändert werden kann. Um C-Strukturen darstellen zu können, kann mit Hilfe des Attributs *pointer* angegeben werden, dass das Datenmodell einen Zeiger repräsentiert. Ist dies der Fall muss mittels *pointerDepth* definiert werden, welche Tiefe der Zeiger besitzt. Beispielsweise besitzt der Zeiger void** eine Tiefe von 2.

3.2 Datenelemente

Datenelemente repräsentieren Werte verschiedener Datentypen. Sowohl einfache Typen, wie *Number*, als auch komplexere Typen, wie zum Beispiel *XmlElement* sind verfügbar. Auch die Erstellung eigener Datentypen ist möglich. Alle Datenelemente besitzen ein Attribut *name*, welches das Element identifiziert und so die Referenzierung an anderer Stelle erlaubt. Jedes Datenelement besitzt außerdem Attribute, *minOccurs*, *maxOccurs* und *occurs*, die angeben wie oft das Element vorkommen kann oder muss. Des Weiteren hat jedes Datenelement folgende Attribute *ref*, *constraint*, *mutable*, *pointer* und *pointerDepth*, die den gleichnamigen Attributen im Datenmodell entsprechen und analoge Funktionalität bereitstellen, siehe hierzu 3.1. Ein Datenmodell kann jedes der folgenden Datenelemente beliebig oft enthalten.

- Block
- Number
- String
- Flags
- Flag
- Blob
- Padding

- Choice
- XmlElement
- Custom

3.2.1 String

Das String-Element repräsentiert Zeichenketten und ist auch wie das *s_string* Element in Sulley, das am häufigsten verwendete Element. Analog zu Sulley kann auch für einen Peach-String eine Kodierung angegeben werden, hierzu wird das Attribut *type* genutzt, das folgende Formate unterstützt: ascii, utf16, utf16be, utf32, utf7 und utf8. Auch die Länge des Strings kann durch Verwendung des Attributes *length* definiert werden. Die tatsächliche Länge ergibt sich aus der Kombination des *length*-Attributes und des Attributes *lengthType*, welches die Einheit der Länge des Strings definiert. Valide Werte sind *bits*, *bytes* und *chars*. Das Attribut *nullTerminated* gibt an, ob die Zeichenkette durch das Zeichen \x00 beendet wird. Auch kann der String auf die angegebene Länge aufgefüllt werden, indem mittels des Attributes *padCharacter* festgelegt wird, welches Zeichen zum Auffüllen benutzt werden soll. Valide Werte sind hier Python-Escape-Sequenzen wie \xNN, \r, \n und so weiter.

```
<String name="string1", value="Hello world!" />
<String name="string2", value="Hello world! nullTerminated="true" />
```

Codeauflistung 19: String

3.2.2 Number

Zahlen werden in Peach durch das *Number*-Element repräsentiert. Im Gegensatz zu Sulley, gibt es keine speziellen Zahlentypen für unterschiedliche Längen, stattdessen besitzt das Element ein Attribut *size* mit dem die Länge der Zahl definiert werden kann. Die Länge der Zahl wird in Bit angegeben und kann zwischen 1 und 64 betragen. Das Attribut *signed* gibt, analog zu Sulley, an, ob die Zahl vorzeichenbehaftet ist oder nicht. Auch ist es möglich,

wie bei Sulley, die Byte-Reihenfolge der Zahl zu definieren, hierfür steht das Attribut *endian* zur Verfügung, wobei neben Big Endian (*little*) und Little Endian (*big*) auch die Option Network-Byte-Order (*network*) verfügbar ist. Es ist außerdem möglich zu definieren, wie der Zahlenwert während des Testlaufs zu interpretieren ist. Hierzu dient das Attribut *valueType*, valide Werte sind *string* und *hex*.

```
<Number name="ushort1" value="200" valueType="string" size="16" signed="false" />
<Number name="bigEndianUnsignedHexInt" value="AB CD" valueType="hex"
       size="32" signed="false" endian="big" />
```

Codeauflistung 20: Number

3.2.3 Block

Ein *Block* besteht aus einer Sequenz von Datenelementen. Zu den Datenelementen die in einem Block vorkommen können, zählt auch der *Block*-Datentyp selbst. Dies erlaubt die Modellierung von hierarchisch geschachtelten Datenstrukturen, wie sie zum Beispiel in objektorientierten Sprachen gemeinhin vorkommen.

```
<Block name="parentBlock">
  <Block name="childBlock1">
    <Block name="grandChildBlock1">
      <String value="1" />
    </Block>
    <Block name="grandChildBlock2">
      <String value="2" />
    </Block>
    <String value="3" />
  </Block>
  <String value="4" />
</Block>
```

Codeauflistung 21: Block

Ein Block kann durch ein *Padding*-Element, siehe 3.2.8, ergänzt werden, dass dazu dient einen Block fixer Länge zu erzeugen. Damit erkannt werden kann, ob der Block mit dem im *Padding* definierten Pad-Zeichen aufgefüllt werden muss, muss die Länge des Blocks mittels *length* definiert werden. Falls

kein Padding notwendig ist, kann über *lengthType* angegeben werden wie die Länge des Blocks berechnet werden soll, valide Werte sind hier, analog zu 3.2.1, *bits*, *bytes* und *chars*.

3.2.4 Choice

Das Choice-Element enthält selbst Datenelemente und sonstige Elemente und erlaubt die Definition der Auswahl eines oder mehrerer Elemente aus dieser Menge. Sowohl die Syntax als auch die Semantik ist analog zum Choice-Element das in XML-Schemata verfügbar ist [21]. Wie viele Elemente ausgewählt werden müssen beziehungsweise ausgewählt werden können, bestimmen die Attribute *min_ occurs* und *max_ occurs*.

```
<Choice name="choiceBlock">
  <String name="choice1" value="1">
  <Block name="choice2">
    <String name="choice2Element1" value="1">
    <String name="choice2Element2" value="1">
  </Block>
  <Number name="choice3" value=3/>
</Choice>
```

Codeauflistung 22: Choice

3.2.5 XmlElement

Durch das Datenelement *XmlElement* lassen sich XML-Elemente darstellen. *XmlElement* kann beliebig viele *XmlAttribute*-Datenlemente und ein weiteres Datenelement, zum Beispiel ein weiteres XmlElement, enthalten. Der Name des XML-Elements wird über das Attribut *elementName* angegeben. Mit Hilfe des optionalen Attributs *ns* kann der XML-Namespace des Elements definiert werden.

```
<XmlElement name="example" elementName="Foo">
<XmlElement elementName="Bar">
   <String value="Hello World!" />
</XmlElement>
```

Codeauflistung 23: Xml-Element [13]

3.2.6 XmlAttribute

Durch das Datenelement *XmlAttribute* können XML-Attribute dargestellt werden. Das Attribut *attributeName* dient zur Angabe des Namens des XML-Attributes. Analog zum *XmlElement* kann mittels des optionalen Attributs *ns* der XML-Namespace des XML-Attributes definiert werden.

```
<XmlAttribute attributeName="myAttribute">
       <String value="attributeValue" />
</XmlAttribute>

<XmlAttribute attributeName="myOtherAttribute ns="someNamespace">
       <String value="attributeValue" />
</XmlAttribute>
```

Codeauflistung 24: Xml-Attribut

3.2.7 Blob

Mit Hilfe des *Blob*-Datenelements können Binärdaten als Hexadezimalwert modelliert werden. Analog zum Datenelement *String* kann die Länge eines Blobs über das Attribut *length* definiert werden. Wieder ergibt sich tatsächliche Länge aus der Kombination des *length*-Attributes und des Attributes *lengthType*. Die möglichen Werte für *lengthType* sind wiederum *bits*, *bytes* und *chars*. Mittels des Parameters *signed* kann analog zum Number-Element angegeben werden, ob der Blob vorzeichenbehaftet ist.

```
<Blob name="blob1" valueType="hex" value="11 c3 64 27 ef"/>
```

Codeauflistung 25: Blob

3.2.8 Padding

Das Padding-Element dient dazu ein Datenmodell oder einen Block an einer Byte-Grenze auszurichten. Die Daten des Modells oder Blocks werden mit dem im Padding-Element angegebenen Wert bis zur nächsten Byte-Grenze aufgefüllt. Die Größe der Grenze wird mit dem Attribut *alignment* angegeben und mittels des Attributs *alignedTo* wird das Datenelement definiert, auf

dessen Basis die Ausrichtung erfolgen soll.

```
<DataModel name="PaddingExample">
       <String name="someString" />
       <Padding aligned="true" alignedTo="someString" alignment=32/>
</DataModel>
```

Codeauflistung 26: Padding

3.2.9 Flags und Flag

Das Datenelement *Flags* erlaubt die Definition eines Feldes von Kennzeichenbits (Flags). Mit dem Attribut *endian* wird die Byte-Reihenfolge definiert, wie beim Datenelement *Number* sind hier wieder die Optionen *little*, *big* und *network* verfügbar. Die Länge des Bitfeldes muss über das Attribut *size* angegeben werden, valide Längen sind 8, 16, 32 und 64. Wird das Boolean-Attribut *padding* auf *true* gesetzt, werden die Werte des Bitfeldes vom Parser gelesen wie ein C-Struct. Hierbei werden immer Padding-Bits eingefügt, wenn ein Flag die Variablengrenze überschreitet beziehungsweise die Größe des folgenden Flags ungleich des aktuellen Flags ist (siehe hierzu [4]). Das optionale Boolean-Attribut *rightToLeft* wird nur berücksichtigt, falls Padding aktiviert wurde. Wird es aktiviert, das heißt der Wert wird auf *true* gesetzt, werden die Bits anstatt von links nach rechts von rechts nach links interpretiert. Das Datenelement *Flags* kann neben den sonstigen Datenelementen, siehe 3.2.10, nur *Flag*-Elemente enthalten. Das *Flag*-Element erlaubt die Definition einer Menge von Kennzeichenbits. Das Attribut *position* legt die Startposition des Flags im übergeordneten *Flags*-Element fest. Die Länge des Flags wird durch das Attribut *size* definiert.

```
<Flags name="options" size="16">
  <Flag name="compression" position="0" size="1" />
  <Flag name="compressionType" position="1" size="3" />
  <Flag name="opcode" position="10" size="2" value="5" />
</Flags>
```

Codeauflistung 27: Flags und Flag [8]

Es gilt das die Summe aus Position und Länge nicht größer sein darf als die Länge des übergeordneten *Flags*-Elements.

3.2.10 Sonstige Elemente

Neben Datenelementen stellt Peach weitere Elemente bereit, die entweder in einem Datenmodell enthalten sein können oder mit einem Datenmodell verknüpft werden können. Diese Elemente ermöglichen es Daten zu laden, das Rendern der Daten zu beeinflussen oder Beziehungen zu anderen Datenmodellen oder -elementen zu modellieren. Folgende Elemente sind verfügbar.

- Data
- Relation
- Transformer
- Fixup
- Hint
- Placement

3.2.10.1 Data Um Standarddaten vorzudefinieren und vorhalten zu können, verfügt Peach über das Element *Data*. Als Datenquelle für das *Data*-Element kann einerseits eine Datei mittels des Attributs *fileName* definiert werden.

```
<Data name="HelloWorldDataSet">
  <Field name="FooBlock.Value" value="Hello World!" />
</Data>

<Data name="LoadFromFile" fileName="sample.bin" />

<Data name="LoadFromExpression" expression="myFuncGetData()" />
```

Codeauflistung 28: Beispiele von Data-Elementen [5]

Des Weiteren kann auch eine Python-Funktionen, die über das Attribut *expression* referenziert werden kann, als Datenquelle genutzt werden. Schließlich ist es auch noch möglich einzelne Elemente eines Datenmodells gezielt

mit Werten zu füllen, hierzu existiert das Subelement *Field*, dessen *name*-Attribut zum Verweis auf das entsprechende Element innerhalb eines Datenmodells dient.

Im Gegensatz zu Datenmodellen, werden *Data*-Elemente nicht schon beim Parsen des Peach-Pits ausgewertet, sondern erst beim Start eines Tests. Damit die Daten in ein Datenmodell geladen werden, müssen das Datenmodell und das *Data*-Element in einem *Action*-Element eines Zustandsmodells (= State Model) referenziert werden. Das Zustandsmodell muss wiederum im ausgeführten Test referenziert werden.

```
<DataModel name="Foo">
   <Block name="Bar">
         <String name="Value" />
   </Block>
</DataModel>

<Data name="HelloWorldData">
       <Field name="Bar.Value" value="Hello world!" />
</Data>

<StateModel name="TheStateModel" initialState="Initial">
  <State name="Initial">
       <Action type="output">
              <DataModel ref="Foo"/>
              <Data ref="HelloWorldData" />
       </Action>
  </State>
</StateModel>

<Test name="Default">
       <StateModel ref="TheStateModel" />
       <Publisher class="Console"/>
</Test>
```

Codeauflistung 29: Beispiel der Verwendung eines Data-Elements

Im Beispiel wird bei Ausführung des Tests *Default* dem String-Element *Value* des Blocks *Bar* enthalten im Datenmodell *Foo* der Wert 'Hello world!' zugewiesen.

```
0000h: 48 65 6C 6C 6F 20 77 6F 72 6C 64 21          Hello world!
```

Abbildung 1: Data: Gerenderte Daten

3.2.10.2 Relation Ein Datenmodell kann beliebig viele Elemente vom Typ *Relation* enthalten. Mit Hilfe des Relationselements ist es möglich Beziehungen zwischen Datenelementen zu modellieren. Typische Beziehungen sind "X ist die Größe von Y", "X ist die Anzahl von Y", or "X ist der Offset (in Bytes) von Y"[11]. Diese Relationstypen sind auch bereits im Peach-Framework implementiert, namentlich *Size-of Relation*, *Count-of Relation* und *Offset-of Relation*.

Size-of ist der am häufigsten verwendete Relationstyp in Peach. Dies liegt im Wesentlichen daran, das C-Structs oft ein Längenfeld enthalten, das beispielsweise die Länge eines enthaltenen Arrays angibt.

```
<DataModel name="TheDataModel">
	<Number size="32" signed="false">
		<Relation type="size" of="Value" />
	</Number>
	<String name="Value" value="Hello world!"/>
</DataModel>
```

Codeauflistung 30: Beispiel einer Size-of Relation [11]

Das vorhergehende Beispiel einer Relation des Typs *Size-of* resultiert in folgenden Binärdaten.

```
0000h: 0C 00 00 00 48 65 6C 6C 6F 20 77 6F 72 6C 64 21  ....Hello world!
```

Abbildung 2: Relation: Gerenderte Daten

Nachfolgend ein Beispiel für Deklaration einer *Count-of* Relation.

```
<DataModel name="TheDataModel">
	<Number size="32" signed="false">
		<Relation type="count" of="Strings" />
	</Number>
	<String name="Strings" nullTerminated="true" maxOccurs="1024" />
</DataModel>
```

Codeauflistung 31: Beispiel einer Count-of Relation [11]

Die *Offset-of* Relation erlaubt die Modellierung von Formaten die eine dynamische Anpassung von Offsets erfordern [11].

```
<DataModel name="TheDataModel">
	<String length="4" padCharacter=" ">
		<Relation type="offset" of="Offset0" />
	</String>
	<String length="4" padCharacter=" ">
		<Relation type="offset" of="Offset1" />
	</String>
	<String length="4" padCharacter=" ">
		<Relation type="offset" of="Offset2" />
	</String>
	<String length="4" padCharacter=" ">
		<Relation type="offset" of="Offset3" />
	</String>
	<String length="4" padCharacter=" ">
		<Relation type="offset" of="Offset4" />
	</String>
	<String length="4" padCharacter=" ">
		<Relation type="offset" of="Offset5" />
	</String>
	<String length="4" padCharacter=" ">
		<Relation type="offset" of="Offset6" />
	</String>
	<Block>
		<Block name="Offset0">
			<Block>
				<String name="Offset1" value="CRAZY STRING!" />
				<String value="aslkjalskdjas" />
				<String value="aslkdjalskdjasdkjasdlkjasd" />
			</Block>
			<String name="Offset2" value="ALSKJDALKSJD" />
			<Block>
				<String name="Offset3" value="1" />
				<String name="Offset4" value="" />
				<String name="Offset5" value="1293812093" />
			</Block>
		</Block>
	</Block>
	<String name="Offset6" value="aslkdjalskdjas" />
</DataModel>
```

Codeauflistung 32: Beispiel einer Offset-of Relation [11]

3.2.10.3 Transformer Einem Datenmodell kann ein *Transformer* zugewiesen werden. Transformer dienen dazu Daten des beinhaltenden Elements zu kodieren und zu dekodieren. Typische Beispiele für solche Kodierungen sind ZIP-Kompression, Base64-Kodierung und kryptographische Verfahren,

wie AES. Peach stellt bereits eine Reihe von Transformern zur Verfügung, siehe [6].

```
<DataModel name="TheDataModel">
	<String name="Value" type="utf8" value="Hello G&#252;nter!"/>
	<Transformer class="UrlEncode"/>
</DataModel>
```

Codeauflistung 33: Beispiel eines Transformers

Die Anwendung des Transformers *UrlEncode* im obigen Beispiel resultiert in der folgenden Binärdarstellung.

```
0000h: 48 65 6C 6C 6F 2B 47 25 63 33 25 62 63 6E 74 65 Hello+G%c3%bcnte
0010h: 72 21                                           r!
```

Abbildung 3: Transformer: Gerenderte Daten

3.2.10.4 Fixup Ein *Fixup*-Element arbeitet, im Gegensatz zu einem *Transformer*-Element, auf Basis der Daten die aus einem *anderen* Datenelement generiert werden. Beispiele hierfür sind Prüfsummen, wie CRC32, und Hashes, wie SHA1. Peach stellt bereits einige vordefinierte Fixups bereit, siehe [7]. Durch Hinzufügen eines *Param*-Elements und setzen des Attributs *ref* auf den Namen des gewünschten Fixups kann dieser referenziert werden.

```
<DataModel name="TheDataModel">
	<String name="value" value="Hello World" />
	<Blob>
		<Fixup class="SHA1Fixup">
			<Param name="ref" value="value" />
		</Fixup>
	</Blob>
</DataModel>
```

Codeauflistung 34: Beispiel eines Fixups [7]

Obiges Beispiel generiert den String "Hello World", gefolgt von seinem SHA1-Hash.

3.2.10.5 Hint Ein Datenmodell kann beliebig viele Elemente des Typs *Hint* enthalten. Hints liefern zusätzliche Informationen interpretiert werden

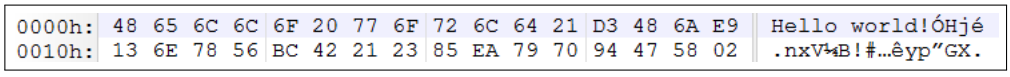

Abbildung 4: Fixup: Gerenderte Daten

können. Für eine detaillierte Beschreibung von Mutatoren und Hints siehe 3.3.

3.2.10.6 Placement Wird einem Datenelement A ein *Placement*-Element zugeordnet, kann die Position des Datenelements auch noch nach dem Parsen des Eingabestreams geändert werden [12]. Im Attribut *before* beziehungsweise *after* wird der Name des Datenelements P angegeben, das als Bezugspunkt für die Platzierung dient. Die exakte Platzierung wird durch eine Relation bestimmt, die im Datenelement P enthalten ist und sich auf Element A bezieht.

```
<DataModel name="Foo">
	<Number name='NumPackets' size='8' >
		<Relation type='count' of='Packets'/>
	</Number>
	<Block name='Wrapper'>
		<Block name='Packets' maxOccurs='1024'>
			<Number name='PacketLength' size='8'>
				<Relation type='size' of='Packet'/>
			</Number>
			<String name='Packet'>
				<Placement after='Wrapper'/>
			</String>
		</Block>
	</Block>
</DataModel>

<Data name="Bar" fileName="placement.bin" />

<StateModel name="TheStateModel" initialState="Initial">
  <State name="Initial">
	<Action type="output">
		<DataModel ref="Foo"/>
		<Data ref="Bar"/>
	</Action>
  </State>
</StateModel>

<Test name="Default">
```

```
        <StateModel ref="TheStateModel" />
        <Publisher class="Console"/>
</Test>
```

Codeauflistung 35: Beispiel eines Placements [12]

Im obigen Beispiel wird der Inhalt der Datei placement.bin, *x020507 !fuzzerpeach*, eingelesen und in das Datenmodell *Foo* gecrackt 1.3.5. Konkret heißt dies, das der Wert *x02* dem Element *NumPackets* zugeordnet wird. Dies bedeutet, dass die Daten genau zwei *Packet*-Elemente und somit auch zwei *PacketLength*-Elemente enthalten. Die Werte *x05* und *x07* werden jeweils einem *PacketLength*-Element zugeordnet und geben die Länge des verknüpften *Packet*-Elements an. Somit werden dem ersten Packet-Element die ersten fünf Buchstaben und dem zweiten Packet-Element die letzten sieben Buchstaben des Strings *!fuzzerpeach* zugewiesen.

```
0000h: 02 05 07 21 66 75 7A 7A 65 72 70 65 61 63 68    ...!fuzzerpeach
```

Abbildung 5: Placement: Gecrackte Daten

3.3 Mutator

Mutatoren dienen dazu den Wert eines Datenmodells oder Datenelements während des Testlaufs zu modifizieren oder durch andere Werte zu ersetzen. Ein Mutator führt genau eine bestimmte Art von Datenmutation durch. Der *NumericalVarianceMutator* generiert beispielsweise Zahlen in einem bestimmten Wertebereich, indem er den Wert eines Datenelements bis zu einem bestimmten Grenzwert verringert und erhöht.

```
[Mutator("Produce numbers that are defaultValue - N to defaultValue + N")]
[Hint("NumericalVarianceMutator-N", "Gets N by checking node for hint, or returns
    default (50).")]
public class NumericalVarianceMutator : Mutator
{
    // ...
}
```

Codeauflistung 36: Beispiel eines Mutators

Wiederum stellt Peach hier bereits einige Implementierungen zur Verfügung, siehe [9].

3.3.1 Hints

Jedes Peach-Datenelement kann eine Liste von sogenannten *Hints* enthalten, die dazu dienen zusätzliche Informationen für die Datenmutation zu liefern. Hierbei wird die Art des Wertes über den eigentlich Typ hinaus spezifiziert. Jeder Hint besitzt eine *name*-Eigenschaft, die in jedem Fall gesetzt werden muss, damit im Mutator der entsprechenden Hint aus der Liste des Elementes ausgewählt werden kann. Des Weiteren besitzt ein Hint noch eine Eigenschaft namens *value*, die mit einem Wert belegt werden kann, der bei Aufruf des Mutators als Parameter übergeben wird.

Der Wert eines String-Elements kann beispielsweise aus einem String bestehen, der ausschließlich numerische Zeichen enthält. In diesem Fall kann das Element mit dem Hint *NumericalString* ausgestattet werden. Somit können beim Testlauf für dieses Datenelement sowohl String- als auch numerische Mutatoren angewandt werden.

```
<String value="250">
   <Hint name="NumericalString" value="true" />
</String>
```

Codeauflistung 37: Verwendung eines Hints

3.3.2 Custom

Peach Version 2.3 enthielt einen Datentyp *Custom*, der das Erstellen eigener Datenelemente ermöglichte. Hierzu musste der *class*-Eigenschaft des *Custom*-Elements die Instanz einer Klasse übergeben werden, die das Parsen, das Cracking und die Generierung der Daten implementiert. In Peach Version 3 ist dieses Feature aktuell (noch) nicht implementiert, sodass zu diesem Zeitpunkt noch keine Informationen über die entsprechende Implementierung vorliegen.

4 Vergleich und Transformation

In folgendem Abschnitt werden die statischen und dynamischen Strukturen der Datenmodelle der beiden Frameworks miteinander verglichen und es wird erläutert wie Datenstrukturen von einem Framework in das andere überführt werden können.

4.1 Zeichenketten (strings)

In beiden Frameworks ähneln sich die beiden verwendeten string-Typen sehr, daher ist die Transformation möglich, ohne dass Erweiterungen am Quellcode eines oder beider Frameworks notwendig sind. Zusätzliche Eigenschaften wie *Kodierung* und *Padding* haben jeweils eine Entsprechung im anderen Framework, siehe 2.

Tabelle 2: Strings

Fuzzer	Objekte	Kodierung	Padding	Terminierung
Sulley	string	encoding	size + padding	✗
Peach	String	type	length + padCharacter	nullTerminated

Allerdings unterscheidet sich die Menge der verfügbaren Kodierungen für Strings zwischen den beiden Frameworks. Während Sulley alle Kodierungen verwenden kann, die in Python verfügbar sind, unterstützt Peach nur die Kodierungen die in Tabelle 3 aufgelistet sind.

Tabelle 3: Gemeinsame Kodierungen

Sulley	Peach
ascii	ascii
utf_16	utf16
utf_16_be	utf16be
utf_32	utf32
utf_7	utf7
utf_8	utf8

4.1.1 Transformation von Strings von Sulley nach Peach

Ein Sulley-String kann leicht in einen Peach-String umgewandelt werden, vorausgesetzt der Sulley-String liegt in einer Kodierung vor, die von Peach unterstützt wird. Ist dies der Fall, kann die Transformation durch einfaches Mapping der Klassen-Members des Sulley-Strings auf die Member des Peach-Strings erfolgen. Liegt der Sulley-String in einer Kodierung vor die Peach nicht beherrscht, wie zum Beispiel *ISO-8859-1*, wäre eine Erweiterung des Peach-Frameworks, um die entsprechende Kodierung nötig. Dies ist mit begrenztem Aufwand zu realisieren, da .NET einen ähnlichen Umfang an Kodierungen bereitstellt wir Python. Das heißt die Kodierungfunktionen selbst werden bereits vom .NET-Framework bereitgestellt, somit muss Peach nur dahingehend erweitert werden, dass es diese Kodierungen benutzen kann. Dies kann durch die Erweiterung der *StringType*-Enumeration erreicht werden. Eine weiterer Unterschied von Peach-Strings gegenüber Sulley-Strings ist die Berechnung der Länge (Peach: *length*) beziehungsweise Größe (Sulley: *size*). Peach erlaubt es verschiedene Einheiten für die Länge anzugeben (*bits*, *bytes*, *chars*). Für die Transformation bedeutet dies, dass die Länge des Peach-Strings auf die Einheit *bytes* umgerechnet werden muss, da dies die einzige Längeneinheit ist, die für Sulley-String zur Verfügung steht.

4.1.2 Transformation von Strings von Peach nach Sulley

Die Transformation eines Peach- in einen Sulley-*strings* kann ebenfalls durch ein einfaches Mapping der Eigenschaften erfolgen. Den einzigen Sonderfall stellen hierbei Peach-Strings dar, die mit dem Nullzeichen abgeschlossen (null-terminated) werden. Um diese Strings korrekt darzustellen, muss nach dem eigentlichen Sulley-String noch ein *binary* Element mit dem Nullzeichen angefügt werden.

```
s_string(...)
s_binary('\x00')
```

Codeauflistung 38: Null-terminated

4.2 Zahlen

Wie die String-Elemente sind auch die Zahlen-Elemente in den beiden Frameworks sehr ähnlich umgesetzt, siehe 4. Sowohl in Sulley als auch in Peach besitzen Zahlenelemente eine bestimmte Länge/Größe. Ebenso kann in beiden Frameworks für Zahlen-Elemente die Byte-Reihenfolge spezifiziert werden. Auch die Angabe ob ein Zahlen-Element vorzeichenbehaftet sein soll, kann ebenso in Sulley als auch Peach vorgenommen werden.

Tabelle 4: Zahlen

Fuzzer	Objekte	Größe/Länge	Bytereihenfolge	Vorzeichen
Sulley	byte word dword qword	1 Byte 2 Byte 4 Byte 8 Byte	endian (big, little)	signed
Peach	Number	1 bis 64 Bit	endian (big, little, network)	signed

4.2.1 Transformation von Zahlen von Sulley nach Peach

Sulley verfügt über mehrere Zahlen-Elemente, die sich nur in ihrer Länge unterscheiden (siehe 2.2.2). Bei der Transformation in eine Peach-Number kann die Länge/Größe somit direkt aus dem Sulley-Elementtyp übernommen werden. Ein eventuell vorhandenes Vorzeichen und die Byte-Reihenfolge können ebenfalls durch einfaches Mapping überführt werden.

4.2.2 Transformation von Zahlen von Peach nach Sulley

Die Umwandlung von Peach-Zahlen-Elementen zu Sulley-Zahlen-Elementen ist ebenfalls durch einfaches Mapping möglich. Sonderfall sind Peach-Zahlen-Elemente die eine Größe/Länge ungleich 1, 2, 4, oder 8 besitzen. Hierfür kann in Sulley der Datentyp *bit_field* genutzt werden. Dieser wird im Abschnitt 4.3 näher erläutert.

Ein weiteres Hindernis bei der Umwandlung können Peach-Zahlen-Elemente darstellen, die die Byte-Reihenfolge *network* verwenden, da dieser Endian-

Wert in Sulley nicht verfügbar ist. Um ein solches Datenelement korrekt transformieren zu können ist es notwendig die Byte-Reihenfolge des untersuchten Netzwerk-Protokolls zu kennen. Beispielsweise ist die Byte-Reihenfolge des IP-Protokolls *Big-Endian* [17], wohingegen die Byte-Reihenfolge des Ethernet Powerlink Protokolls *Little-Endian* ist [15].

4.3 Binärdaten

Beide Frameworks verfügen über jeweils zwei unterschiedliche Datentypen, um binäre Daten zu modellieren. Peach verfügt über die Datentypen *Number* und *Blob* zur Darstellung binärer Daten, wohingegen Sulley für diesen Zweck die Datentypen *bit_field* und *static* besitzt.

Sulley bietet den Typ *bit_field* an der es erlaubt binäre Daten beliebiger Länge darzustellen. Da *bit_field* die Basisklasse der verschiedenen Sulley-Zahlentypen, wie beispielsweise *byte* oder *dword*, ist, verfügt dieser Typ auch über die selben Eigenschaften, wie zum Beispiel das Zielformat *format* oder die Byte-Reihenfolge *endian*. Zusätzlich besitzt *bit_field* die Eigenschaft *size* mit der die Größe der Binärdaten angegeben werden kann. Somit können mit Hilfe dieses Typs auch Zahlen dargestellt werden, deren Länge von der der Standard-Zahlentypen abweicht (analog zu *Number*) in Peach. Allerdings ist die Länge des *bit_fields*, im Gegensatz zum Peach *Number*-Datentyp, in der Länge unbegrenzt.

Peach stellt zur Modellierung binärer Daten den Datentyp *Blob* bereit. Es ist allerdings anzumerken, dass sich dieser Datentyp vom Sulley-Datentyp *bit_field* insofern unterscheidet, dass es mit dem Typ *Blob* nicht möglich ist, die Byte-Reihenfolge zu definieren. Der Nutzer muss also sicherstellen, dass die binären Daten in der korrekten (das heißt zum modellierten Protokoll passenden) Byte-Reihenfolge vorliegen. Ähnlich zu *bit_field* kann auch für den Peach-*Blob* ein Format über das Attribut *valueType* definiert werden. Die verfügbaren Werte sind hier *hex*, *string* und *literal*. Die beiden ersteren Werte sind hierbei analog zu den Sulley-Formaten *binary* beziehungsweise *ascii*. Wird der *valueType* allerdings auf *literal* gesetzt, wird als Wert des Attributes *value* ein Python- oder Ruby-Ausdruck erwartet, dessen Ergeb-

nis dann direkt als Binärdaten interpretiert wird. Diese Funktionalität ist allerdings in der aktuellen Peach-Version (3.0.207) noch nicht implementiert. Auch der Datentyp *Number* kann zur Darstellung binärer Daten verwendet werden. Allerdings können hiermit nur binäre Daten bis zur einer Länge von 64 bit dargestellt werden.

Tabelle 5: Binärdaten

Fuzzer	Objekte	Bytereihenfolge	Vorzeichen
Sulley	bit_field	endian	signed
Peach	Blob	✗	✗

4.3.1 Transformation von Binärdaten von Sulley nach Peach

Als Ziel-Datentyp für die Konvertierung eines Sulley-bit_fields bietet sich, trotz oben erwähnter Unterschiede, der Peach-Datentyp *Blob* an. Allerdings müssen die Binärdaten vor der Transformation gegebenenfalls in die gewünschte Byte-Reihenfolge konvertiert werden, da diese in einem Peach-Blob nicht definiert werden kann. Ist die Länge des *bit_fields* kleiner als 64 bit, kann als Zieldatentyp auch der Peach-Typ *Number* verwendet werden. Da dieser es auch erlaubt die Byte-Reihenfolge zu definieren, sollte diese Konvertierung, unter Berücksichtigung der oben genannten Voraussetzung, bevorzugt werden.

4.3.2 Transformation von Binärdaten von Peach nach Sulley

Zur Konvertierung eines Peach-*Blobs* in ein Sulley-*bit_field* ist es ebenfalls notwendig, die Byte-Reihenfolge des Blobs zu kennen. Ist dies der Fall, kann die Konvertierung durch einfache Übernahme des Wertes erfolgen. Soll eine Peach-*Number* in ein Sulley-*bit_field* umgewandelt werden, ist die Byte-Reihenfolge bekannt und die Konvertierung kann ohne weitere Voraussetzungen durchgeführt werden.

4.4 Blöcke

Da beide Frameworks einen block-basierten Ansatz verfolgen, ist natürlich von elementarer Bedeutung, dass auch Blöcke ineinander konvertiert werden können. Da beide Blocktypen eine Sammlung von Datenelementen darstellen, muss die Struktur des Blocks bei der Konvertierung nicht verändert werden. Wie in den Codeauflistungen 6 und 7 zu sehen ist, hat ein Großteil der Zusatzeigenschaften der Blöcke eine Entsprechung im jeweils anderen Framework. Somit können auch diese Eigenschaften transformiert werden.

Tabelle 6: Blockeigenschaften 1

Fuzzer	Objekt	Kodierung	Größe des Blocks	Padding
Sulley	block	encoder	size	✗
Peach	Block	Transformer	sizeOf-Relation	Padding

Tabelle 7: Blockeigenschaften 2

Fuzzer	Objekt	Prüfsumme / Hash	Bedingung	Wiederholung
Sulley	block	checksum	dep dep_value(s) dep_compare	repeat
Peach	Block	Fixup	✗	min_occurs max_occurs

4.4.1 Transformation von Blöcken von Peach nach Sulley

Der wesentliche Teil der Transformation eines Peach-Blocks zu einem Sulley-Block ist die Konvertierung der enthaltenen Datenelemente. Da beide Blocktypen selbst auch wieder Blöcke enthalten können, muss dies bei der Konvertierung berücksichtigt werden, indem Kind-Blöcke rekursiv transformiert werden. Ist für den Peach-Block ein *Transformer* angegeben, der den Block in eine bestimmte Kodierung überführt, kann diese Kodierung auch für den transformierten Sulley-Block angegeben werden. Hierzu muss eine Python-Funktion existieren, die den Kodierungsalgorithmus des Peach-Transformers

implementiert. Ist dies der Fall kann diese Funktion über die Eigenschaft *encoder* des Sulley-Blocks referenziert werden. Die Umwandlung eines *Fixups* ist analog zur Umwandlung eines Transformers, nur muss hier der Verweis auf die entsprechende Python-Funktion innerhalb eines *checksum*-Elementes angegeben werden, dass wiederum auf den transformierten Block verweist. Die *min_ occurs* und *max_ occurs*-Eigenschaften des Peach-Blocks können durch Hinzufügen eines Sulley-*Repeat*-Elements mit der Schrittweite (step) 1 und Übernahme der *occurs*-Werte in *min_ reps* beziehungsweise *max_ reps*. Existiert eine *sizeOf-Relation* die sich auf den Peach-Block bezieht, kann diese durch Hinzufügen eines entsprechenden Sulley-*size*-Elements mit Bezug auf den transformierten Sulley-Block ebenfalls konvertiert werden. Verfügt der Peach-Block über ein *Padding*-Element, kann dieses nicht direkt transformiert werden. Denkbar wäre es die Padding-Funktionalität in eine Python-Funktion zu kapseln, die dann gegebenenfalls bei der Kodierung des Blocks ausgeführt wird.

4.4.2 Transformation von Blöcken von Sulley nach Peach

Auch bei der Umwandlung eines Sulley-Blocks in einen Peach-Block müssen wieder die im Block enthaltenen Datenelemente konvertiert werden. Wiederum müssen Kind-Blöcke rekursiv transformiert werden. Ist die *encoder*-Eigenschaft des Sulley-Blocks gesetzt und existiert ein *Transformer* für Peach der dieselbe Kodierung implementiert, kann die Kodierung in den transformierten Block übernommen werden. Hierzu muss der resultierende Peach-Block auf den entsprechenden Transformer verweisen. Zur Umwandlung eines mit dem Sulley-Block verknüpften *checksum*-Elements gilt wieder die Voraussetzung, dass das für Peach ein *Fixup* existieren muss, der ebenfalls den, durch das *checksum*-Element definierten Prüfsummen- oder Hash-Algorithmus, implementiert. Wiederum muss der resultierende Peach-Block auf das entsprechende *Fixup*-Element verweisen. Mit dem Sulley-Block verknüpfte *size*-Elemente können durch Hinzufügen eines entsprechenden Peach-*Number*-Elements, das über eine *sizeOf*-Realtion mit dem resultierenden Peach-Block verknüpft ist, umgewandelt werden. Sulley-*repeat*-Elemente kön-

nen ebenfalls transformiert werden, indem die Werte von *min_ reps* beziehungsweise *max_ reps* in die entsprechenden Eigenschaften, *min_ occurs* respektive *max_ occurs*, des resultierenden Peach-Blocks übernommen werden. Lediglich die Schrittweite (step) des repeat-Elements kann hier nicht mit umgewandelt werden. Zu den Eigenschaften, *dep*, *dep_ value(s)* und *dep_ compare* existiert kein Gegenstück im Peach-Framework, deshalb ist eine Erweiterung des Peach-Frameworks notwendig, um eine 1:1 Umwandlung möglich zu machen. Die Erweiterung von Peach um eine *when*-Relation wäre eine denkbare Lösung, sie könnte dazu dienen einen Block (oder ein beliebiges Datenelement) aus einem *Choice*-Element abhängig von einer Bedingung auszuwählen. Eine solche Erweiterung würde allerdings umfangreiche Änderungen am Peach-Framework erfordern, sodass im Rahmen dieser Arbeit darauf verzichtet wird.

4.5 Benutzerdefinierte Typen

Da Peach in der aktuellen Version keine Definition benutzerdefinierter Typen erlaubt, ist eine Konvertierung eines Sulley-legos nur möglich, falls dieser einem vorhandenen Peach-Elementtyp gleicht (wie zum Beispiel *tag* und *XmlElement*) oder sich durch vorhandene Peach-Elemente nach-modellieren lässt. Da dies nur in speziellen, einfachen Fällen möglich ist und eine automatisierte Transformation beliebiger, benutzerdefinierten Typen (selbst bei Unterstützung von benutzerdefinierten Typen durch Peach) eine hohe Komplexität impliziert, wird die Transformation benutzerdefinierter Typen an dieser Stelle nicht weiter ausgeführt.

4.6 Peach-spezifische Elemente

Folgender Abschnitt beschreibt die Transformation von Peach-spezifischen Datenelementen in Sulley-Datenelemente. Da in Sulley für diese Elemente kein Gegenstück existiert, müssen die Peach-Elemente entweder durch Komposition aus primitiven Sulley-Elementen oder Erweiterungen des Sulley-Frameworks modelliert werden. Dabei ist es nicht in allen Fällen möglich,

alle syntaktischen und semantischen Eigenschaften automatisiert zu transformieren.

4.6.1 Flags- und Flag-Elemente

Flags und *Flag*-Elemente stellen eine Sonderform von Binärdaten dar, daher ist der Sulley-Datentyp *bit_field* am besten als Zieltyp geeignet. Ein dynamisches Umordnen der Flag-Elemente, wie es in Peach unter Verwendung der *Position*-Eigenschaft möglich ist, wird von Sulley durch keinen Datentyp unterstützt. Dies kann somit nur durch eine Erweiterung des Zielframeworks, in diesem Fall von *bit_field*, erreicht werden.

4.6.2 Xml-Datentypen

Peach stellt, wie in den Abschnitten 3.2.5 und 3.2.6 beschrieben, spezielle Datenelemente für Xml-Datentypen bereit. Da Sulley keine solchen Datentypen besitzt, müssen die Peach-Xml-Datentypen durch primitive Sulley-Datentypen nachmodelliert werden.

Xml-Element Um ein Xml-Element zu transformieren werden die einzelnen Bestandteile des Xml-Elements (Klammern, Tag-Name, Leerzeichen vor Attributen) durch Sulley-*delim*- und *static*-Elemente dargestellt. Falls das Xml-Element selbst wieder Xml-Elemente enthält, müssen diese durch rekursives Ausführen der Transformation konvertiert werden. Ist der Wert hingegen vom Typ *String* oder *Number*, kann der Wert eines Xml-Elements direkt von Peach nach Sulley konvertiert werden.

Xml-Attribut Zur Transformation eines Xml-Attributs muss dieses, analog zum Xml-Element, aus primitiven Sulley-Elementtypen, nach-modelliert werden.

4.6.3 Choice

Sulley hat kein zu Choice vergleichbares Datenelement, deshalb ist eine einfache Konvertierung durch Mapping nicht möglich. Eine Möglichkeit ein sol-

ches Element nach Peach zu konvertieren ist es für jedes Kind-Element das im Choice-Element enthalten ist ein eigenes Datenmodell (in Sulley) zu erstellen. Zur Veranschaulichung soll ein kurzes Beispiel dienen. In der Code-Auflistung 39 ist das Ausgangsmodell dargestellt, es enthält ein String- und ein Choice-Element. Die Möglichkeiten die sich aus der Kombination der beiden Elemente ergeben sind: „Hello world!“, „Hello Sulley!“ und „Hello Peach!“.

```
<DataModel name="DataModelWithChoice">
  <String value="Hello "/>
  <Choice name="Choice">
    <String value="world!"/>
    <String value="Sulley!"/>
    <String value="Peach!"/>
  </Choice>
</DataModel>
```

Codeauflistung 39: Beispiel eines Datenmodells mit Choice

Um diese drei Varianten nach Sulley zu übernehmen, werden drei Datenmodelle generiert, siehe 40.

```
s_initialize('DataModel1')
s_string('Hello ')
s_string('world!')

s_initialize('DataModel2')
s_string('Hello ')
s_string('Sulley!')

s_initialize('DataModel3')
s_string('Hello ')
s_string('Peach!')
```

Codeauflistung 40: Transformation von Choice nach Sulley

Diese Transformation ist zwar denkbar einfach, hat aber den Nachteil das sie statisch ist. Statisch bedeutet in diesem Zusammenhang, dass nur jeweils eins der drei Datenmodelle aus dem Beispiel während des Test zur Verwendung kommt, je nachdem welches Datenmodell für den Test referenziert wurde. Um alle drei Datenmodelle zu testen, wäre also ein dreimaliger Testdurchlauf nötig und zwar jeweils mit einem anderen Datenmodell. In einfachen Testszenarien mag dies eine legitime Vorgehensweise darstellen,

in umfangreicheren Szenarien kann die Erhöhung der Testdauer allerdings inakzeptable Ausmaße annehmen. Daher ist eine alternative Methode zur Umwandlung wünschenswert, die auch die Laufzeiteigenschaften des Choice-Elements konserviert.

Ein Spezialfall tritt ein, wenn das *Choice*-Element ausschließlich *String*-Elemente enthält, dann kann das Choice-Element in ein Sulley-*Group*-Element umgewandelt werden. Dies hat den Vorteil, dass sich das Group-Element während eines Testlaufs dynamisch verhält, dass heißt sein Wert wird automatisch vom Sulley-Framework mutiert.

4.6.4 Padding

Eine direkte Konvertierung von *Padding*-Elementen von Peach nach Sulley ist nur möglich, falls der Block oder das Datenmodell neben dem Padding-Element nur noch ein weiteres String-Element enthält. In diesem Fall kann die *padding*-Eigenschaft des Sulley-*string*-Typs zur Transformation genutzt werden. Die *padding*-Eigenschaft des Sulley-*string*-Elementes wird mit dem Wert des Peach-*Padding*-Elementes belegt und die *size*-Eigenschaft des Sulley-*strings* wird auf den *alignment*-Wert des Padding-Elementes gesetzt. Diese Transformation ist allerdings nur valide, falls der String nicht ein Vielfaches der vorgegeben Länge erreichen kann, da die Größe Sulley-strings beim Auffüllen als absolute Grenze interpretiert wird. Ist obige Voraussetzung nicht erfüllt, das heißt das Datenmodell oder der Block enthält also neben dem Padding-Element weitere Elemente (nicht nur ein weiteres *string*-Element), ist eine direkte Formatierung des *Padding*-Elements nicht möglich. Der Grund hierfür ist, dass ein Sulley-*block*-Element keine für Padding vorgesehene Eigenschaft besitzt. Allerdings ist es durch Nutzung der *encoder*-Eigenschaft eines Sulley-*block*-Elements möglich, das Verhalten eines Peach-*Padding*-elementes zu nachzubilden. Dies kann erreicht werden, indem eine Kodierungsfunktion genutzt wird, die die Daten auf eine bestimmte Länge auffüllt anstatt die Daten umzukodieren.

4.6.5 Sonstige Peach-Elemente

Die sonstigen Elemente die Peach bereitstellt können nur zum Teil nach Sulley konvertiert werden. Insbesondere Elemente die auf *Data Cracking* zurückgreifen sind kaum konvertierbar, da Sulley keine diesbezüglichen Mechanismen enthält.

4.6.5.1 Data Elemente vom Typ *Data* sind generell nicht nach Sulley konvertierbar, da Sulley keine Infrastruktur für die Zerlegung von Binärdaten bereitstellt. Eine dynamische Integration des Data Cracking in Sulley würde eine Reimplementierung der entsprechenden Strukturen von Peach erfordern und wäre daher sehr aufwändig. Im konkreten Einzelfall ist es natürlich möglich Sulley um Funktionen zu erweitern, die es erlauben bekannte Datenstrukturen zu zerlegen, oder andere Programmierbibliotheken zu nutzen die diese Funktionalität bereitstellen.

4.6.5.2 Relation Von den drei verschiedene Relationstypen in Peach, *Size-of*, *Offset-of* und *Count-of*, kann nur die *Size-of* Relation nach Sulley transformiert werden. Hierzu kann der Sulley Block-Hilfstyp *sizer* genutzt werden. Die Nutzung eines Block-Hilfstypen indiziert, dass diese Funktionalität nur für Sulley-Blöcke verfügbar ist. Dies bedeutet, dass die Elemente nach der Umwandlung in einen Sulley-Block eingebettet werden müssen. Somit ist das Resultat der Umwandlung des in Codeauflistung 30 dargestellten Peach-Datenmodells wie folgt.

```
s_initialize("DataModel")
# length (in Bytes) entspricht length (in Bits)
s_size("block1", length=4, signed=False)
if s_block_start("block1"):
    s_string("Hello world!")
s_block_end()
```

Codeauflistung 41: Peach sizeOf-Relation zu Sulley-Block mit Sizer

Die anderen Relationstypen können mangels entsprechender Funktionalität nicht ohne signifikante Erweiterungen des Sulley-Frameworks korrekt transformiert werden.

4.6.5.3 Transformer Grundvoraussetzung zur Umwandlung eines Elements vom Typ *Transformer* ist, dass die entsprechenden Funktionen zur Kodierung und Dekodierung auch in Sulley verfügbar sind. Entweder sind die entsprechenden Funktionen bereits im Sulley-Framework enthalten, werden von einer Dritt-Bibliothek bereitgestellt oder müssen neu implementiert werden. Unter obiger Voraussetzung können Elemente des Typs *Transformer* direkt nach Sulley umgewandelt werden, wenn sie in einem *Block*-Element enthalten sind. Dies ist möglich, da der *Block*-Typ in Sulley die Möglichkeit bietet einen Encoder zu definieren, der das semantische Äquivalent zum *Transformer* in Peach darstellt. Ist das *Transformer*-Element in einem anderen Datenelement enthalten, muss zuerst das enthaltende Element nach Sulley transformiert werden. Danach wird das transformierte Element in einen Sulley-Block eingebettet und dessen *encoder*-Eigenschaft gesetzt. Da der Block keine anderen Elemente enthält, ist sichergestellt, das die Kodierungsfunktion das selbe Ergebnis liefert wie bei einer Kodierung des Elements selbst. Das Peach-Datenmodell aus der Codeauflistung 33 kann nach Sulley transformiert werden, falls auf Seiten des Sulley-Frameworks eine Funktion zur Url-Kodierung eines String vorhanden ist. Unter der Annahme, dass diese Kodierungsfunktion existiert (diese sei im Folgenden als *url_ encode* bezeichnet) ist das Resultat der Transformation des Beispiel-Datenmodells wie in Codeauflistung 42 dargestellt.

```
s_initialize("TheDataModel")
if s_block_start("DummyBlock" encoder=url_encode):
        s_string("Hello World")
s_block_end()
```

Codeauflistung 42: Ergebnis der Transformation eines Datenelementes mit Transformer

4.6.5.4 Fixup Peach-*Fixup*-Elemente können unter Nutzung des *checksum*-Blockelementes transformiert werden. Hierzu müssen die Elemente wie bei 4.6.5.3 einen Sulley-Block eingebettet werden. Die nachfolgende Codeauflistung zeigt das Ergebnis der Umwandlung des Datenmodells aus Codeauflistung 34.

```
s_initialize("TheDataModel")
if s_block_start("DummyBlock"):
        s_string("Hello World")
s_block_end()
s_checksum("DummyBlock", algorithm="sha1")
```

Codeauflistung 43: Ergebnis der Transformation eines Datenelementes mit Fixup

4.6.5.5 Hint Die Konvertierung von *Hints* ist generell nicht möglich, da Sulley kein Konzept für Mutatoren besitzt. Allerdings kann die *fuzz_ library*-Eigenschaft des jeweiligen Elements mit zum aktuellen Szenario passenden Werten belegt werden. Somit ist eine Anpassung der generierten Werte auf Subtypen des Element-Datentyps möglich.

4.6.5.6 Placement *Placement*-Elemente können im Allgemeinen nicht von Peach nach Sulley konvertiert werden, da Sulley keine Meachanismen zum Data Cracking vorsieht.

4.7 Sulley-spezifische Elemente

Im folgenden Abschnitt werden die Transformation der Elemente die ausschließlich im Sulley-Framework vorhanden sind beschrieben. Im Gegensatz zur Umwandlung von Peach-spezifischen Elementen kann in den meisten Fällen eine direkte Umwandlung durchgeführt werden oder es sind nur marginale Änderungen am Peach-Framework nötig, um eine direkte Konvertierung zu ermöglichen.

4.7.1 Statischen Datentypen

Peach besitzt, anders als Sulley, kein spezielles Datenelement für Strings oder andere Typen die nicht mutiert werden sollen. Wird das *mutable*-Attribut eines Peach-Strings oder einer Peach-Number auf *false* gesetzt, wird der Wert des jeweiligen Elementes während des Testlaufs nicht verändert. Auf diese Weise lässt sich ein *static*-Object nach Peach transformieren.

4.7.2 Trennzeichen

Im Gegensatz zu Sulley besitzt Peach keinen speziellen Datenelementtyp für Trennzeichen. Der signifikante Unterschied zwischen s_string und s_delim, beziehungsweise dem durch den Aufruf von s_delim erstellten Objekt der Klasse *delim*, ist die Mutation der Werte (siehe 2.2.1 und Codeauflistung 44).

```
# Wiederhole Trennzeichen einige Male,
# falls es nicht leer ist.
if self.value:
        self.fuzz_library.append(self.value * 2)
        self.fuzz_library.append(self.value * 5)
        self.fuzz_library.append(self.value * 10)
        self.fuzz_library.append(self.value * 25)
        self.fuzz_library.append(self.value * 100)
        self.fuzz_library.append(self.value * 500)
        self.fuzz_library.append(self.value * 1000)

# Versuche das Trennzeichen auszulassen.
self.fuzz_library.append("")

# Falls das Trennzeichen das Leerzeichen ist,
# fuege ein paar Tabs hinzu.
if self.value == " ":
        self.fuzz_library.append("\t")
        self.fuzz_library.append("\t" * 2)
        self.fuzz_library.append("\t" * 100)

# Fuege weitere Standard-Trennzeichen hinzu.
self.fuzz_library.append(" ")
self.fuzz_library.append("\t")

# ...
```

Codeauflistung 44: delim

Peach erlaubt eigene Mutatoren zu definieren, daher ist es möglich einen eigenen *DelimiterMutator* für Peach-Strings zu entwickeln, der dieselbe Mutationslogik verwendet wie das s_delim-Element in Sulley. Hierzu genügt das Ableiten der Klasse *StringMutator*, die Teil der Peach-API ist, und ein Befüllen des enthaltenen String-Arrays *values* mit den Werten, die die Klasse *delim* in ihrem internen Array *fuzz_library* hält (siehe Codeauflistung 45).

```
public partial class DelimMutator : StringMutator
{
    public void generateValues(DataElement obj)
    {
        if (supportedDataElement(obj))
        {
            string value = (string)((Dom.String)obj).DefaultValue;
            List<string> valueList = new List<string>();
            // # if the default delim is not blank, repeat it a bunch of times.
            if (value != null && value != string.Empty)
            {
                valueList.Add(repeatString(value, 2));
                valueList.Add(repeatString(value, 5));
                valueList.Add(repeatString(value, 10));
                valueList.Add(repeatString(value, 25));
                valueList.Add(repeatString(value, 100));
                valueList.Add(repeatString(value, 500));
                valueList.Add(repeatString(value, 1000));
            }
            // try ommitting the delimiter.
            valueList.Add(string.Empty);
            // the delimiter is a space, try throwing out some tabs
            if (value == " ")
            {
                valueList.Add("\t");
                valueList.Add(repeatString("\t", 2));
                valueList.Add(repeatString("\t", 100));
            }
            // toss in some other common delimiters:
            valueList.Add(" ");
            valueList.Add("\t");
            valueList.Add(repeatString("\t ", 100));
            valueList.Add(repeatString("\t\r\n", 100));
            valueList.Add("!");
            // ...
            values = valueList.ToArray();
        }
        else
        {
            throw new ArgumentException(string.Format("DataElement of type '{0}' not
                supported.", obj.GetType().FullName));
        }
    }
}
```

Codeauflistung 45: DelimMutator

4.7.3 Random

Um Sulley-*random_data*-Elemente nach Peach konvertieren zu können, ist es wie bei Trennzeichen (siehe Abschnitt 4.7.2) nötig einen speziellen Mutator zu erstellen. Dieser muss den in Sulley verwendeten Mutationsalgorithmus implementieren. Da Peach-Elemente nicht die Eigenschaften *min_length*, *max_length* und *num_mutations* besitzen (siehe 2.2.4), müssen diese Informationen mit Hilfe eines entsprechenden Hints *Random* (siehe Codeauflistung 46) an den Mutator übergeben werden. Das *value*-Attribut des Hints enthält hierzu eine komma-separierte Liste von Integern, deren Elemente die Werte für die oben genannten Eigenschaften darstellen.

```
<DataModel name="TheDataModel">
       <String name="random" value="1111">
              <Hint name="Random" value="4,6,50"/>
       </String>
</DataModel>
```

Codeauflistung 46: Hint für Random-Mutation

Die Implementierung des Mutators ist unter Verwendung des Random-Hints dann trivial, da nur noch der Python-Quelltext der *mutate*-Methode des Sulley-random_data-Elements in C# re-implementiert werden muss.

5 Implementierung

Die Implementierung des Konvertierungsprogramms wurde in C# (.NET 4.0) durchgeführt. Zusätzlich wurde auch das IronPython-Framework genutzt, um Sulley-Protokollspezifikationen die als Python-Skript vorliegen, einlesen zu können.

5.1 Converter.Common

Um die Datenelemente der beiden untersuchten Frameworks auf eine gemeinsamen Klassenstruktur abbilden zu können, wurde eine Basisschicht namens *Converter.Common* erstellt. Sie enthält Schnittstellendefinitionen für die verschiedenen Datenelemente. Die zentrale Schnittstelle der Basisschicht ist *IDataField<T>*, welche selbst von den Schnittstellen *IDataField* beziehungsweise *INamed* abgeleitet ist (siehe Abbildung 6).

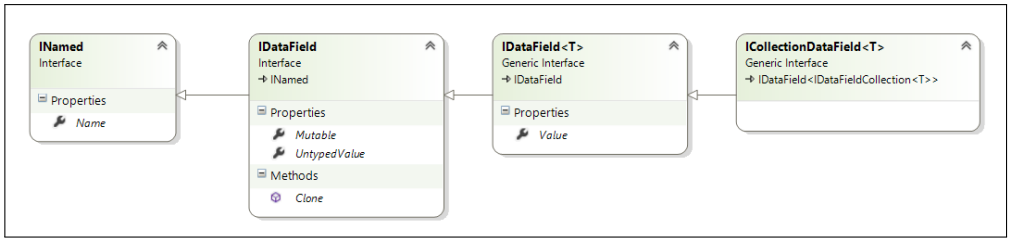

Abbildung 6: IDataField

IDataField<T> besitzt die Eigenschaften *Name*, *Mutable*, *UntypedValue* und *Value*. Die Eigenschaft *Name* dient dazu den Namen des Datenelementes zu speichern. Die bool-Eigenschaft *Mutable* gibt die Veränderlichkeit des Wertes des Elements während des Testlaufs an. Die Eigenschaft *UntypedValue* ist zur Speicherung Wertes des Datenelementes vorgesehen, wohingegen die Eigenschaft *Value* den in UntypedValue gespeicherten Wert typisiert, das heißt als Instanz vom Typ T, zurückliefert. Für Elemente deren Typ nicht skalar sondern eine Sammlung von weiteren Datenelementen ist (zum Beispiel der Typ *Block*) existiert die Schnittstelle *ICollectionDataField<T>*, die von IDataField<T> abgeleitet ist.

Des Weiteren enthält die Basisschicht die Standardimplementierungen der

oben genannten Schnittstellen. Die Klasse *DataField<T>* ist die zentrale Klasse der Basisschicht und ist die Standardimplementierung der Schnittstelle IDataField<T>. Die Klasse *CollectionDataField<U>* ist abgeleitet von DataField<T> und ist die Standardimplementierung der Schnittstelle ICollectionDataField<T>.

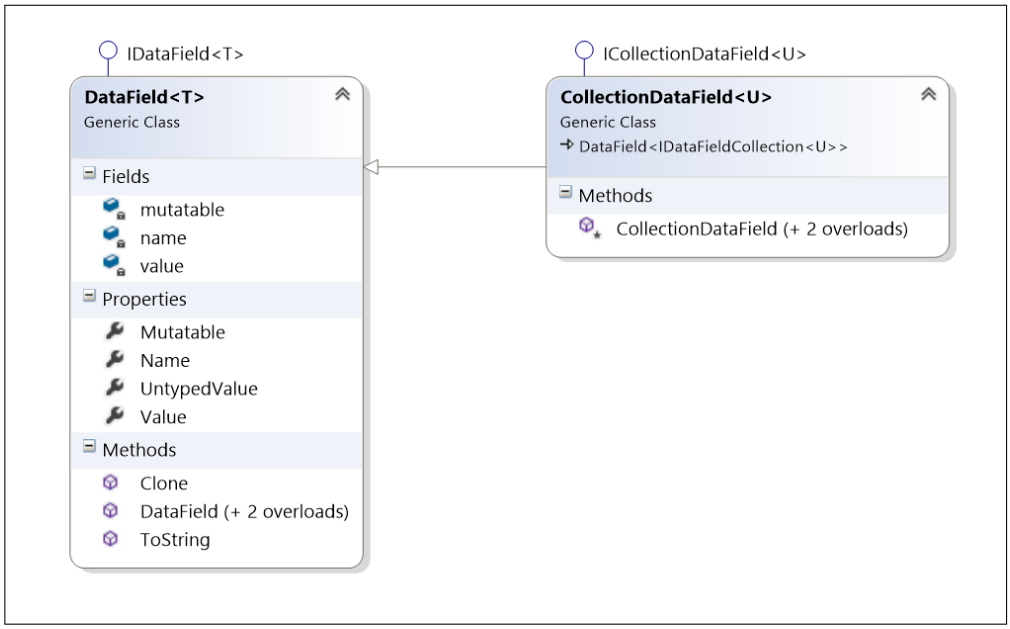

Abbildung 7: DataField

Alle Datenelemente mit skalarem Werttyp, wie String, Integer oder BitField (siehe Abbildung 8) sind direkt von DataField<T> abgeleitet.

Datenelemente deren Wert aus eine Liste von Datenelementen besteht, wie beispielsweise der Elementtyp Block (siehe Abbildung 9), erben stattdessen von der Klasse *CollectionDataField*.

Da Datenelemente Bestandteil eines Datenmodells sind, existieren auch hierfür jeweils eine Schnittstelle *IDataModel<T>* und eine Standardimplementierung *DataModel*, siehe Abbildung 10.

5.2 Converter.Peach3

Auf Basis der oben beschriebenen Basisschicht wurde eine Abstraktionsschicht für Peach3-Datenelemente implementiert. Alle Peach3-Elemente sind

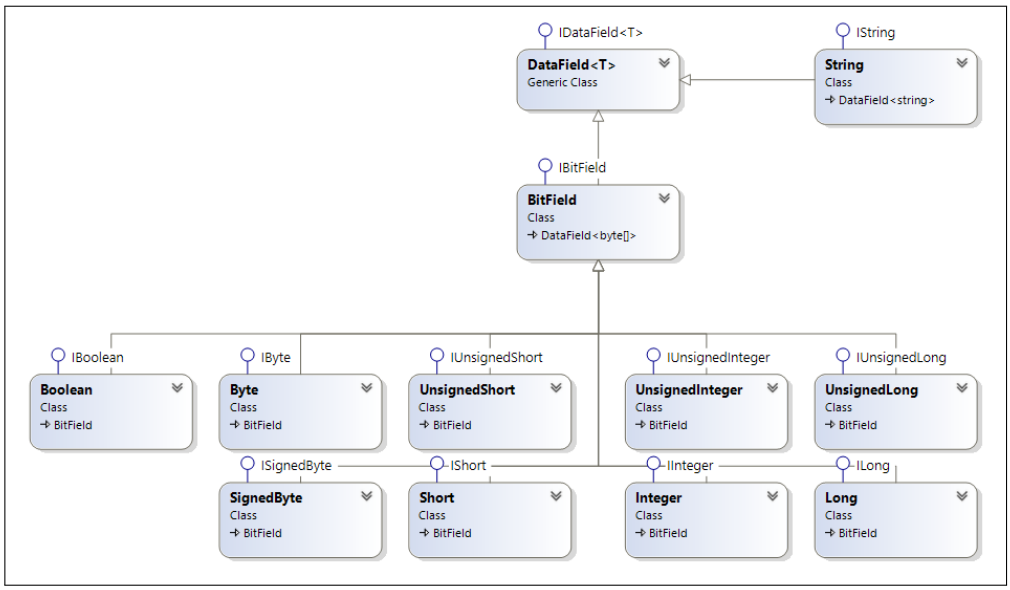

Abbildung 8: DataField-Klassen

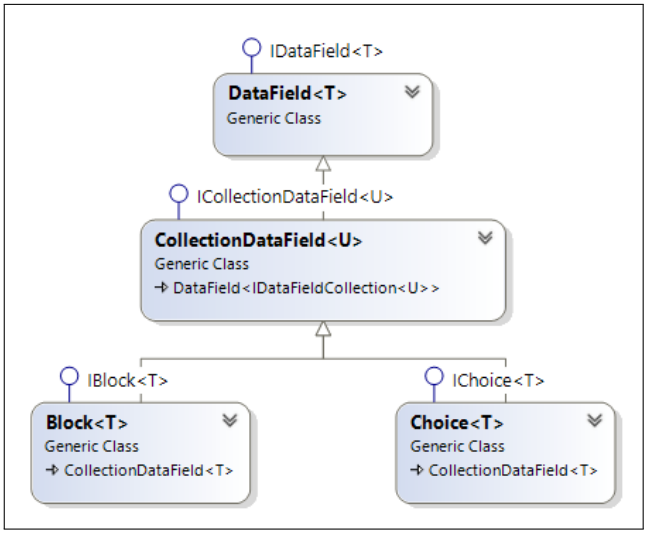

Abbildung 9: CollectionDataField-Klassen

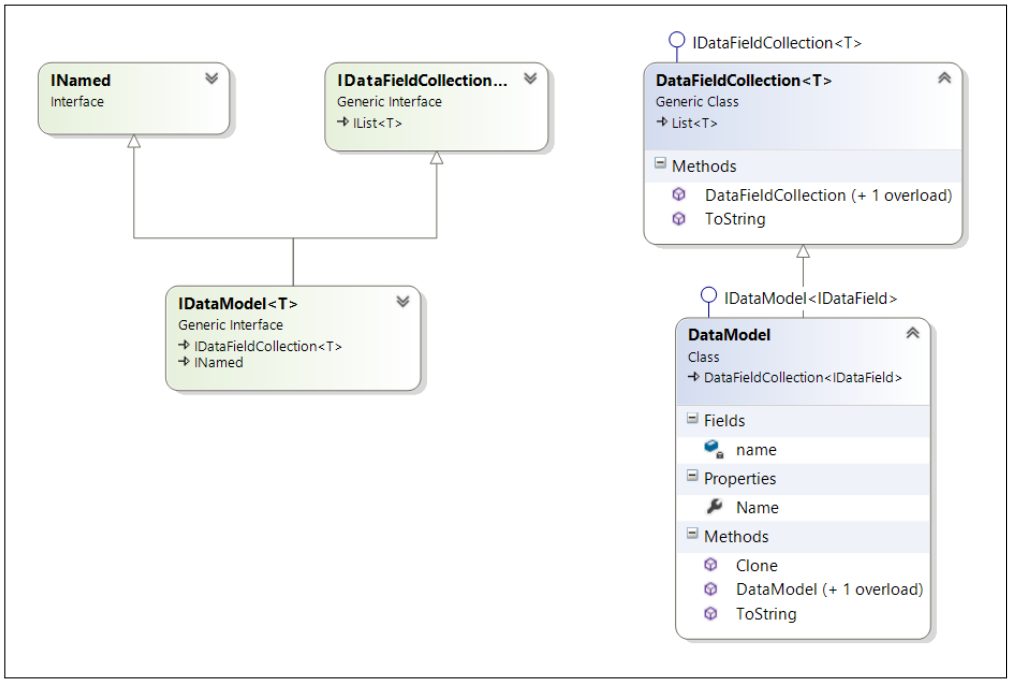

Abbildung 10: DataModel

von einem entsprechenden Element in der Common-Schicht abgeleitet. Zusätzlich enthält jedes Peach3-Element eine Methode namens *Serialize*, die das Datenelement in Peach3-XML serialisiert.

5.3 Converter.Sulley

Die Implementierung der Sulley-Abstraktionsschicht ist analog zur Abstraktionsschicht für Peach3-Elemente, auch hier erbt jedes Datenelement von einem Element aus der Common-Schicht. Ebenfalls besitzt jedes Sulley-Datenelement eine Methode *Serialize* die aus den Eigenschaften des Elementes den entsprechenden Python-Ausdruck generiert.

5.4 Converter.Transform

Der Namensraum *Converter.Transform* enthält die statische Klasse *Transformer*, welche wiederum sechs statische Subklassen enthält. Jede der enthaltenen Subklassen implementiert die Methode *TransformDataModels*, welche

eine Collection von Datenmodellen der Ausgangschicht als Parameter erwartet und eine Collection von Datenmodellen der Zielschicht zurückgibt.

- SulleyToSulleyLayer
- SulleyLayerToCommon
- CommonToSulleyLayer
- PeachToPeachLayer
- PeachLayerToCommon
- CommonToPeachLayer

Da die zugrundeliegenden Frameworks, Sulley und Peach, keine Möglichkeit anbieten Datenmodelle und ihre Elemente in das Spezifikationsformat (.xml oder .py) zu serialisieren, ist eine Konvertierung von Common zum Ausgangsframework nicht sinnvoll. Deshalb enthalten die Datenelemente der Abstraktionsschichten entsprechende Methoden zur Serialisierung.

5.5 Converter.ViewModel

Das ViewModel dient zur Anbindung der in den Abstraktionsschichten enthaltenen Informationen in der Benutzeroberfläche. Die Datenelement-Klassen des ViewModels implementieren zwar die Basisinterfaces definiert in Common, teilen sich aber nicht die Implementierung mit den Basisklassen. Stattdessen nimmt der Konstruktor eines ViewModel-Elements eine Common-Instanz eines Datenelementes entgegen und speichert diese in einem eigens dafür definiertem Feld. Diese Architektur erlaubt es sowohl Datenelemente aus der Sulley- als auch aus der Peach3-Abstraktionsschicht an den Konstruktor des ViewModel-Elementes zu übergeben.

6 Zukünftige Arbeiten

Vorliegender Bericht behandelt nur die Datenmodellierung in den Frameworks Peach und Sulley. Der zweite Teil des Berichts wird die Beschreibung von Zustandsmodellen und deren Konvertierung, sowie die Durchführung von Tests und der Analyse der Testergebnisse enthalten. In einer weiteren Arbeit [18] wird die Abstraktion von Zahlen und Binärdaten im Detail erörtert.

7 Fazit

Durch intensive Code-Reviews beider Frameworks und die Entwicklung eines Prototyps zur Konvertierung von Datenmodellen zwischen Sulley und Peach konnten einige Erkenntnisse gewonnen werden, die bei zukünftigen Vergleichen von verschiedenen Datenmodellierungsansätzen hilfreich sind.

Besonders die Unterschiede zwischen Zahl- und Binärdatentypen in beiden Frameworks müssen hierbei hervorgehoben werden. Zwar beinhalten beide Frameworks Elemente, um solche Datentypen zu modellieren, allerdings unterscheiden sich die jeweiligen Typen durch ihre Eigenschaften, sodass eine Transformation nur teilweise oder durch Eingabe des Benutzers möglich ist.

Peach stellt außerdem auch einige komplexere Datentypen, wie *XmlElement*, *Choice* oder *Padding*, bereit, die in Sulley nicht verfügbar sind. Die Umwandlung solcher Typen ist ohne Erweiterung des Zielframeworks nur bedingt durchführbar, da sich nicht nur die Strukturen unterscheidet, sondern auch das Verhalten während des Testlaufs von einfacheren Datentypen abweicht. Die Datenstruktur kann insofern transformiert werden, dass die einzelnen Elemente durch ein oder mehrere primitive Elemente des Zielframeworks dargestellt werden.

8 Quellen

Literatur

[1] D. Aitel. *SPIKE*. 2004. URL: http://www.immunitysec.com/downloads/SPIKE2.9.tgz.

[2] P. Amini. *PaiMei*. 2012. URL: https://github.com/OpenRCE/paimei.

[3] P. et al. Amini. *OpenRCE / Sulley*. 2013. URL: https://github.com/OpenRCE/sulley.

[4] *Bitfields*. Sweepscape. 2013. URL: http://www.sweetscape.com/010editor/manual/Bitfields.htm (besucht am 28. 04. 2013).

[5] M. Eddington. *Data*. 2013. URL: http://peachfuzzer.com/v3/Data.html.

[6] M. Eddington. *Default Transformers*. 2013. URL: http://peachfuzzer.com/v3/Transformer.html.

[7] M. Eddington. *Default Transformers*. 2013. URL: http://peachfuzzer.com/v3/Fixup.html.

[8] M. Eddington. *Flags*. 2013. URL: http://peachfuzzer.com/v3/Flags.html.

[9] M. Eddington. *Mutators*. 2013. URL: http://peachfuzzer.com/v3/Mutators.html.

[10] M. Eddington. *Peach fuzzing platform*. 2013. URL: http://peachfuzzer.com.

[11] M. Eddington. *Relation*. 2013. URL: http://peachfuzzer.com/v3/Relation.html.

[12] M. Eddington. *SHAFixup*. 2013. URL: http://peachfuzzer.com/v3/Fixups/SHA1Fixup.html.

[13] M. Eddington. *XmlElement*. 2013. URL: http://peachfuzzer.com/v3/XmlElement.html.

[14] Python Software Foundation. *Python standard encodings*. 2013. URL: http://docs.python.org/2/library/codecs.html#standard-encodings.

[15] Ethernet POWERLINK Standardisation Group. *Ethernet POWERLINK*. Ethernet POWERLINK Standardisation Group. URL: http://www.ethernet-powerlink.org/index.php?id=12&tx_abdownloads_pi1[action]=getviewclickeddownload&tx_abdownloads_pi1[uid]=48&no_cache=1.

[16] A. Pedram und Portnoy A. *Sulley: fuzzing Framework*. OpenRCE.

[17] J. Postel und J.K. Reynolds. *Standard for the transmission of IP datagrams over IEEE 802 networks*. RFC 1042 (Standard). Internet Engineering Task Force, Feb. 1988. URL: http://www.ietf.org/rfc/rfc1042.txt.

[18] F. Schmidt. „Extending models for numbers and binary data in fuzzing-frameworks to improve convertibility“. preprint (2013).

[19] M. Sutton, A. Greene und P. Amini. *Fuzzing. Brute Force Vulnerability Discovery*. Addison-Wesley, 2007, S. 361–364.

[20] A. Takanen, J. D. Demott und C. Miller. *Fuzzing for software security testing and quality assurance*. Mit einem Vorw. von B. Miller. Artech House on Demand, 2008.

[21] H. et al. Thompson. *XML Schema Part 1: Structures Second Edition*. W3C. 2004. URL: http://www.w3.org/TR/2004/REC-xmlschema-1-20041028 (besucht am 28. 10. 2004).