DATA FORMAT DESCRIPTION AND ITS APPLICATIONS IN IT SECURITY

Michael Hartle[1]*, Andreas Fuchs*[2]*, Marcus Ständer*[1]*, Daniel Schumann*[1]*, Max Mühlhäuser*[1]

Telekooperation[1],
Dept. of Computer Science, TU Darmstadt / Germany,
{mhartle, staender, max}@tk.informatik.tu-darmstadt.de

Fraunhofer-Institut SIT[2], Darmstadt / Germany,
andreas.fuchs@sit.fraunhofer.de

ABSTRACT

Data formats play a central role in information processing, exchange and storage. Security-related tasks such as the documentation of exploits or format-aware fuzzing of files depend on formalized data format knowledge. In this article, we present a model for describing arbitrary data format instances as well as arbitrary data formats as a whole. Using the *Bitstream Segment Graph (BSG)* model and the *BSG Reasoning* approach, we describe a PNG image serving as exploit for Adobe Photoshop CS2 (CVE-2007-2365). We furthermore show directions how our work can be applied to secure data format design as well as formal security analysis.

Index Terms— Data format description, finite bit sequences, documentation of exploits, formal security validation

1. INTRODUCTION

1.1. Motivation

The concept of data formats is central to information processing, exchange and storage. A *data format* defines how information is represented as bits, bytes or characters and thus shapes every format-compliant process that *determines syntax and semantics of data* in order to access information. When a bit sequence is transmitted, both sender and receivers need to agree on its data format for interoperability.

1.2. Use Cases

Knowledge regarding syntax and semantics of data is relevant to IT Security as can be shown through several use cases related to data format knowledge:

- *Documentation of exploits* describe the composition of malicious crafted data. Such a documentation helps developers to explore syntax and semantics of malicious data, understand the delivery mechanism of the exploit, and so to fix processing bugs in affected applications.
- The definition of a data format can include security-relevant flaws hidden in the complexity of a specification, which may lead to exploitable implementations. A visual refined representation can aid security experts with the *identification of tripwires* such as redundant information. If applied during during the modelling phase, such flaws may be prevented or at least be limited by adding security-related programming advice to the specification.
- Similarly, good knowledge about utilized data formats is necessary in the area of *formal security validation*, which is among others required for certain Assurance Levels in Common Criteria. Formally based data format descriptions can therefor fullfill the requirements for reasoning about assumptions and abstractions made in this process.

1.3. Problem

For such use cases, a vast, growing number of existing and evolving data formats turns designing, implementing and maintaining format-specific solutions into a repetitive, neverending task. We can improve the situation by separating data format knowledge into reusable *declarative*, *machine-processible* descriptions for *format-agnostic* solutions. Realizing this idea requires formalized models on *data format instances* as well as on *data formats as a whole*, where we define a data format instance as a finite bit sequence and a data format as a (possibly infinite) set of data format instances [1].

Data formats have been subject of research in various domains like *Multimedia* or *Telecommunication*, including related work on describing data formats. In *Digital Preservation*, substantial related work for managing data formats exists like the *Open Archival Information System* (OAIS) [2], data format registries such as *Global Data Format Registry* (GDFR) [3] and *PRONOM* [4], or the *Typed Object Model* (TOM) [5] for managing format-related operations on data.

Yet, existing approaches for formally describing data formats is domain-specific and not applicable for arbitrary data formats in general.

1.4. Contribution

We present two contributions for describing both data format instances as well as data formats in the context of IT Security. The first contribution is the *Bitstream Segment Graph (BSG)* model for describing arbitrary data format instances. The second contribution is the *BSG Reasoning* approach for describing arbitrary data formats. We extend previous publications [1, 6, 7] with an analysis on modelling arbitrary data formats in general, the Apeiron BSG editor as tool support for creating, exploring and manipulating BSG instance descriptions, as well as a RDF/N3-based notation for BSG instances which makes them interchangeable and machine-processible.

1.5. Outline

We start with surveying related work in literature on data format description from the domains of Multimedia, Telecommunication and Grid Computing in Section 2, focusing on their descriptive capabilities. We then abstract and analyse elemental properties of both data formats and their instances. Here, we investigate how to guarantee these properties in formal models in Section 3, showing theoretical limits given by formal language and computational theory. Building on our analysis, we present the BSG model for describing data format instances, and our rule-based BSG Reasoning approach for describing data formats in Section 4. Subsequently, the presented contributions are put in the context of IT Security by giving an example of their application on an PNG image exploit for Adobe Photoshop CS2 based on CVE-2007-2365 in Section 5. In addition, we describe further applications such as flaw detection during design and formal security validation in Section 6 and close with a summary in Section 7.

2. RELATED WORK

2.1. Data Format Description

We present related work on describing data formats from the research domains of *Multimedia*, *Telecommunication* and *Grid Computing*.

Here, our focus is on the descriptive capabilities of each approach regarding *structures*, *transcodes*, *fragments* and *primitives*, as well as handling *functional dependencies* between them. Quite self-explanatory, a structure represents a concatenation of data. A primitive is encoded data which represents some information with defined semantics. We define a transcode as data which is compressed, encrypted or the result of another reversible, information-preserving block transformation. Moreover, we define a fragment as data that is part of a larger composite. Functional dependencies include cases where the length of a data segment is determined by another, or where the value of a data segment depends on a computation of another, as for CRCs.

For describing the composition of data format instances in general, all these descriptive capabilities are required. For example, the PNG image shown in Figure 5 carries transformed and compressed image data which is fragmented in two segments.

2.1.1. Multimedia

Primary motivation for research in Multimedia on data format description is the *standardization of data formats*, and the *adaptation of digital items* for Universal Media Access (UMA). Both the *MPEG 1/2 methodology* and the *MPEG SDL and Flavor/XFlavor* approach belong to the former line of research, whereas the latter contributed the *Bitstream Syntax Description Language* (BSDL). There are further approaches such as *BFlavor and gBFlavor* [8, 9] as recombinations of BSDL and Flavor, but which do not provide significant extentions regarding their descriptive capabilities.

2.1.1.1. MPEG 1/2 methodology

The "MPEG-1/2 methodology" [10] was used for describing data formats in various parts of the MPEG-1 (ISO/IEC 11172) and MPEG-2 (ISO/IEC 13818) standards.

It is a notation similar to C `struct` definitions, used in a tabular form to describe elements of a data structure with the respective name, data type and size in bits. As the methodology does not cover functional dependencies between elements, these are typically described textually.

The methodology is a "visual language" and targets humans as audience. It is intended for static data structures featuring bit granularity description, but does not handle functional dependencies such as variable-length data ocurring for the Exponential Golomb integer encoding, in PNG chunks or in MPEG-4 boxes.

2.1.1.2. MPEG SDL and Flavor/XFlavor

The *Syntactic Description Language* (SDL) was proposed for describing variable-length data in MPEG-4 data formats [11], was included as part of the MPEG-4 Systems and Description Languages (MSDL) [12] and used in the specification of MPEG-4. It later became the *Formal Language for Audio-Visual Object Representation (Flavor)* with its XML-based extension XFlavor [13, 14].

In Flavor code, data structures are described as *classes*, mixing data declarations with procedural flow-control statements. Flavor enables the translation to Java and C++ code by generating appropriate methods for parsing and serialization. The language focuses on H.263 or MPEG-2 Video which do not need data to be decoded during parsing, and therefore lim-

its itself to parsing only without any decoding, circumventing this "high-level context" problem [15].

Flavor code is procedural and targets machine-processing. It allows for bit granularity of description, covers both structures and primitives, and handles simple functional dependencies such as variable-length data structures. It does neither handle complex functional dependencies such as CRCs, nor does it explicitly support transcoded data or fragmentation.

2.1.1.3. Bitstream Syntax Description Language (BSDL)

The *Bitstream Syntax Description Language* (BSDL) addresses the *Universal Media Access* (UMA) vision as part of the MPEG-21 Digital Item Adaptation standard in Multimedia [16]. It allows for the adaptation of digital items to enable timely delivery of and access to multimedia resources in highly heterogenous, resource-contrained environments.

BSDL extends XML Schema for describing binary data. It focuses on so-called *scalable data formats*, where an specific adaptation of a bitstream is generated through computationally simple filtering rather than re-encoding. Where BSDL supports the definition of format-specific schemata, the *generic BSDL* (gBSDL) variant defines a format-agnostic, general-purpose alternative for use on low-resource environments such as mobile phones.

BSDL allows for bit granularity and is declarative. It supports structures and primitives, but does not explicitly cover transcoded data or fragmentations. Functional dependencies are described using XPath and are evaluated during runtime on an in-memory XML Document Object Model (DOM) representation of parsed data which may still be incomplete. As a consequence, adapting digital items using the BSDL reference implementation suffers from performance and scalability issues [8].

2.1.2. Telecommunication

Similar to Multimedia, the primary motivation on data format description in Telecommunication is the *standardization of data formats* for enabling interoperability in communications between different parties. Yet here, the focus is on representing data in a defined way rather than describing arbitrary representations in their own composition.

Related work includes the *External Data Representation* (XDR), the well-known *Abstract Syntax Notation One* (ASN.1) including the *Encoding Control Notation* (ECN) and other approaches such as the *Concrete Syntax Notation 1* (CSN.1) or *Transfer Syntax Notation One* (TSN.1).

2.1.2.1. External Data Representation (XDR)

The External Data Representation (XDR) is currently defined in RFC 4506 [17] and standardizes a language for describing data and its encoding. It has been used in the definition of network protocols Sun Remote Procedure Call (RPC) [18], the Network File System (NFS) [19] and others.

The XDR language somewhat resembles the C programming language, but does not include flow-control statements and is declarative. It intends to provide support for the exchange of messages using common high-level data types rather than describing arbitrary data.

XDR provides support for structures and primitives from a limited set of data types. It does neither provide bit-level granularity nor explicitly support transcoded data or fragmentation. Regarding simple functional dependencies, it supports variable-length data through specific data types.

2.1.2.2. Abstract Syntax Notation One (ASN.1) and Encoding Control Notation (ECN)

The *Abstract Syntax Notation One* (ASN.1) has been defined by the International Telecommunication Union, Telecommunication Standardization Sector (ITU-T) [20] and allows the definition of data models as ASN.1 modules. In combination with encoding rules such the Packed Encoding Rules (PER) [21], an ASN.1 module defines a specific data format. Alternativ encoding rules can be specified through the *Encoding Control Notation* (ECN) [22]. ASN.1 is used in the definition of numerous file formats and network protocols, such as for X.509 security certificates [23] or for H.225 and H.245 messages used in the H.323 video conferencing protocol [24].

ASN.1 allows the definition of an abstract data model in a declarative manner. The design of ASN.1 and its encoding rules follow the separation of Application layer and Presentation layer in the ISO OSI stack, which allows for the renegotation of representations [25]. While the separation of ASN.1 module definition and its encoding makes sense in an interactive scenario where encodings can be renegotiated, it provides no benefit for data formats in general, as the definition of data models and its representation is often fixed in advance and without means for renegotiation.

ASN.1 allows the description of structures and primitives, yet it does not natively cover transcoded data or fragmentation. Quite specific functional dependencies are supported such as for variable-length data.

2.1.2.3. Concrete Syntax Notation 1 (CSN.1)

The Concrete Syntax Notation 1 (CSN.1) is a language used in the definiton of messages for GSM and UMTS communication protocols by the European Telecommunication Standards Institute (ETSI). CSN.1 has a published specification [26] and is informally described in Annex B of ETSI TS 100 939 [27].

CSN.1 is declarative and describes the composition of data on the bit level quite similar to a formal language grammar. Although similar in naming to ASN.1, CSN.1 directly describes the composition of data on the bit level, whereas ASN.1 describes an abstract data model.

CSN.1 provides support for structures, primitives and offers quite extensive support of functional dependencies com-

pared to all other approaches. However, it does not provide explicit support for transcoded data or fragmentation of data.

2.1.2.4. Transfer Syntax Notation One (TSN.1)

The *Transfer Syntax Notation One* (TSN.1) has been specified by the company Protomatics, Inc [28] and is used in their commercial product offerings. It defines a language for describing the composition of data, yet to our knowledge, it has not been used for existing, published specifications of well-known data formats.

Naming of TSN.1 follows the ASN.1 and CSN.1 acronyms, but the language describes data in a procedural way not unlike the Flavor approach from Multimedia, mixing data declarations with flow-control statements.

TSN.1 is intended for machine-processing and is capable of describing structures and primitives including the simple functional dependency of variable-length data. As with others, it does not explicitly handle transcoded data or fragmentation.

2.1.3. Grid Computing

Last but not least, describing existing data sets for exchange of research results is of interest in Grid Computing. It includes approaches such as *Binary Format Description* (BFD) and the *Data Format Description Language* (DFDL).

2.1.3.1. Binary Format Description (BFD)

Binary Format Description (BFD) is an XML-based language intended for describing "arbitrary" data formats of scientific datasets [29]. It was developed under funding of Pacific Northwest National Laboratory (PNNL) in 2000 and was part of the Scientific Annotation Middleware (SAM) project by the U.S. Department of Energy [30].

BFD extends the *Extensible Scientific Interchange Language* (XSIL) with flow-control statements and enables functional dependencies through XPath-based references to be resolved during runtime. It is intended for machine consumption, is procedural and covers both structures and primitives. It does not explicitly handle transcoded data or fragmentation.

2.1.3.2. Data Format Description Language (DFDL)

The *Data Format Description Language* (DFDL) is a format description language with the explicit goal of describing any data format [31]. It is an extension to XML Schema and is currently defined in working draft 032 of its initial version 1.0.

Similar to the BSDL approach, DFDL extends the XML Schema with application-specific annotations. In its current form, DFDL focuses on "hierarchical nested data", providing support for both structures and primitives. It also handles functional dependencies through XPath-based references. Yet, it does not explicitly handle transcoded data or fragmentation.

2.1.4. Summary

The descriptive capabilities of related work presented from Multimedia, Telecommunication and Grid Computing are summarized in Table 1. There are several aspects we observed that are interesting to note:

- Related work focuses on specific aspects of data formats, such as simple adaptation of scalable data formats in Multimedia, or ensuring interoperability through defining data models in Telecommunication, often limiting their descriptive capabilities to these aspects. A thorough analysis on modelling data formats in general is notably absent from all of them.
- The distinction between representing and describing data is sometimes blurred in literature, although both tasks are conceptually different. Representing data can be limited to specific representations of information, whereas describing data requires complete support of arbitrary information representations and thus is more complex to achieve.
- Next to no cross-pollination apparently occurs between domains in this regard, although several approaches across domain boundaries have adopted the use of XML, XML Schema and XPath.

There are sufficient machine-processible, declarative approaches that both handle structures and primitives on bit granularity as well as functional dependencies on a simple level. Yet, explicit support for transcoded data and fragmentations is missing from all these approaches. The former is required for handling compressed, encrypted, or otherwise transformed data, whereas the latter is required for interleaved data typical for multimedia files. These cases can be observed in PNG images or MPEG-4 movies.

3. ANALYSIS

3.1. Abstraction

Let $\mathbb{D}$ denote the set of all data formats, $d \in \mathbb{D}$ denote a data format, α denote a sender and β denote a receiver.

HYPOTHESIS 3.1: The current state-of-the-art on data format description can be improved by considering a data format d as a normative set of lossless information representations for purposes of storage and transmission between a sender α and a receiver β.

Following the hypothesis, the basic usage scenario of a data format shown in Figure 1 can be stated as follows:

- A sender α has an internal representation m_α of information. The sender ensures the validity of m_α with respect to a data format d and maps m_α to an external representation m_d, which is then sent over a channel c.

	MPEG 1/2	(X)Flavor	BSDL	XDR	ASN.1 & ECN	CSN.1	TSN.1	BFD	DFDL
Support of structures	⊠	⊠	⊠	⊠	⊠	⊠	⊠	⊠	⊠
Support of primitives	⊠	⊠	⊠	⊠	⊠	⊠	⊠	⊠	⊠
Support of transcodes	□	□	□	□	□	□	□	□	□
Support of fragments	□	□	□	□	□	□	□	□	□
Machine-processible	□	⊠	⊠	⊠	⊠	⊠	⊠	⊠	⊠
Declarative, not procedural	⊠	□	⊠	⊠	⊠	⊠	□	⊠	⊠

Table 1. Comparison of descriptive capabilities

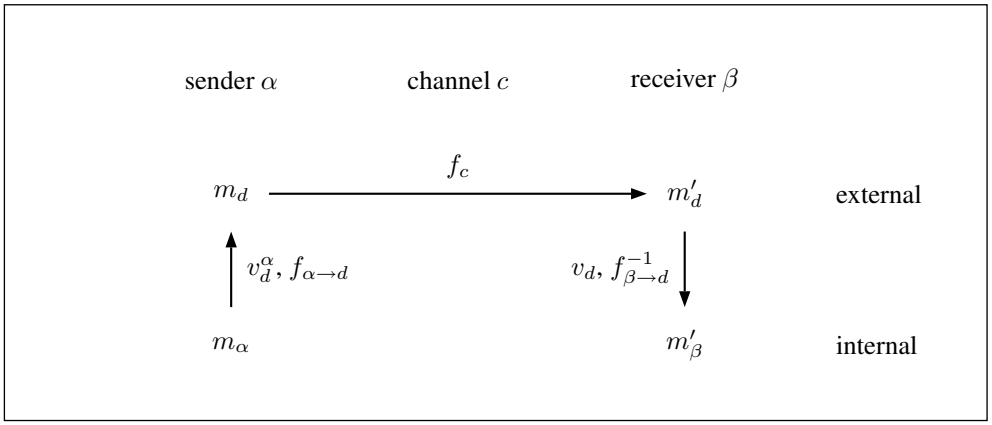

Fig. 1. Abstract information transport using a data format

- A receiver β eventually obtains an external representation m'_d from c, which may be invalid due to an erronous sender, or be different from the originally sent m_d in case of a noisy channel. The receiver therefore ensures the validity of m'_d and maps m'_d to an internal representation m_β.

Both sender and receiver necessarily share information concerning the data format d. Moreover, depending on d, both α and β may share additional context information required for deciding the validity, or for mapping from and to an external representation m_d. Example use cases are the identification of embedded data formats through a separate channel, or the use of encryption in a data format.

3.1.1. Lossless versus lossy data formats

In this regard, it is necessary to clarify the expression "lossless information representation" from our hypothesis with regards to the distinction between lossless and lossy data formats.

A *lossless data format* represents an original information, whereas a *lossy data format* represents an approximation of original information according to a defined metric. In either case, the actually represented information is to be recovered by the receiver, be it original or an approximation, so mappings from and to m_d are necessarily information-preserving. For the course of this article, we therefore define a data format d to specify the representation of information *in a lossless manner*. Aspects regarding the preprocessing of information to be represented through a data format, such as approximations and related metrics for lossy data formats, are not within the scope of this article.

3.2. Formalization

Informally, we define a *data format instance* as a mapping between an internal representation m_γ and an external representation m_d, and define a *data format* through a set of such instances. Formalizing both terms in that order, the following definitions are given step-wise as follows:

3.2.1. Encoding data

As a first step, a way to represent data in an encoded form.

DEFINITION 3.1 (BIT SEQUENCE): A bit sequence b is defined as finite and non-empty. The set of all finite, non-empty bit sequences is defined as $\mathbb{B}$ (Eq. 1).

$$b = \{0,1\}^n, n \geq 1, b \in \mathbb{B} \tag{1}$$

DEFINITION 3.2 (ENCODING): An *encoding* e is a bijective function which maps between an element $x \in \mathbb{X}_e$ and its corresponding bit sequence b (Eq. 2).

$$e : \mathbb{X}_e \leftrightarrow \mathbb{B}_e, \mathbb{B}_e \subseteq \mathbb{B} \tag{2}$$

3.2.2. Representing information

A bit sequence represents encoded *data*, but does not describe its meaning by itself, as it depends on the actual context. In order to represent data including its semantics as *information*, some sort of "labeling" is needed.

DEFINITION 3.3 (LABEL): A *label* l is a symbol that denotes some given semantics. The set of all labels is defined as $\mathbb{L}$.

DEFINITION 3.4 (LABELED BIT SEQUENCE): A *labeled bit sequence* i is defined as a pair $i = (b, \mathbb{L}_i)$, where $b \in \mathbb{B}$ is a bit sequence and $\mathbb{L}_i \subseteq \mathbb{L}$ is a subset of labels that denote the meaning of b (Eq. 3). The set of all labeled bit sequences is defined as $\mathbb{I}$.

$$i = (b, \mathbb{L}_i), b \in \mathbb{B}, \mathbb{L}_i \subseteq \mathbb{L}, i \in \mathbb{I} \tag{3}$$

A labeled bit sequence represents *information* by making the meaning of encoded data explicit regarding the specific context of d. A labeled bit sequence can be categorized depending on whether its information is part of the message to be transported, or whether it is used as "packaging" to enable transportation.

DEFINITION 3.5 (PAYLOAD): A labeled bit sequence $i = (b, \mathbb{L}_i)$ is *payload* if the value of its bit sequence b is functionally independent from other labeled bit sequences.

DEFINITION 3.6 (PACKAGING): A labeled bit sequence $i = (b, \mathbb{L}_i)$ is *packaging* if the value of its bit sequence b is functionally dependent on one or more labeled bit sequences, such as depending on their (relative) location, length, labels or bit sequences.

3.2.3. Representing complex information

Labeled bit sequences serve as building blocks for more complex representations, which are used either as *internal representation* at a sender or receiver γ, or used as *external representation* for exchanging information between a sender and a receiver.

DEFINITION 3.7 (INTERNAL REPRESENTATION): An *internal representation* m_γ represents information in a way that is specific to some sender / receiver γ and is defined as a tuple of one or more labeled bit sequences (Eq. 4) which are semantically distinct. The set of all internal representations is defined as $\mathbb{IR}$.

$$m_\gamma = \{i_1, \ldots, i_n\}, n \geq 1, i_x \in \mathbb{I}, m_\gamma \in \mathbb{IR} \tag{4}$$

Different internal representations may represent the same information, yet in varying *granularity*.

DEFINITION 3.8 (GRANULARITY): The *granularity* of an internal representation m_γ is a relative measure on how fine-grained information is represented. Higher granularity is achieved by a more fine-grained description. The actual granularity of m_γ may vary depending on the sender or receiver γ.

EXAMPLE 3.1: Let $m_{\gamma,1} = \{i\}, i = (b, \mathbb{L}_i)$ be an internal representation, where i represents the color of a pixel as a 24 bit RGB value composed of 8 bits for each color component of red, green and blue. Let $m_{\gamma,2} = \{i_1, i_2, i_3\}, i_x = \{b_x, \mathbb{L}_{i,x}\}$ be an internal representation, where i_1, i_2 and i_3 represent the color of a pixel as a 24 bit RGB value as separate 8 bit red, green and blue color components. In this case, $m_{\gamma,2}$ has a higher granularity than $m_{\gamma,1}$.

Packaging is typically present in a data format in order to describe variable aspects of payload required during the parsing process, such as the length of a variable-length payload. In order to compute packaging information during generation and process packaging information during parsing, a certain minimum granularity of internal representation is required which separates packaging from payload.

DEFINITION 3.9 (EXTERNAL REPRESENTATION): An *external representation* m_d represents information as normatively defined by a data format d. It is defined as a tuple containing exactly one labeled bit sequence (Eq. 5). The set of all external representations is defined as $\mathbb{ER}$.

$$m_d = \{i\}, i \in \mathbb{I}, m_d \in \mathbb{ER} \tag{5}$$

An external representation m_d typically carries some aggregation of information rather than a single primitive value. An external representation m_d is exchanged between sender and receiver over some *channel*.

DEFINITION 3.10 (CHANNEL): An *channel* c passes an external representation $m_d = \{i\}, i = \{b, \mathbb{L}_i\}$ from a sender α to a receiver β, including the bit sequence b and its labels $\mathbb{L}_i$. It is modelled as a channel function f_c (Eq. 6).

$$f_c : \mathbb{ER} \rightarrow \mathbb{ER} \tag{6}$$

A channel c handles the transmission of an external representation $m_d = \{i\}, i = \{b, \mathbb{L}_i\}$ by transferring both the bit sequence b and its labels $\mathbb{L}_i$. A specialized channel c may only pass external representations for a specific set of data formats. Furthermore, a channel c may be noisy and introduce errors into the bit sequence or the set of labels.

3.2.4. Mapping between representations

In order to map between internal and external representations, some means for a bijective *transformation* is needed.

DEFINITION 3.11 (TRANSFORMATION): A *transformation* t is a bijective function which maps between *input* and *output* as two ordered tuples of bit sequences (Eq. 7). As shown in Figure 2, transformations can be categorized through the cardinality of the input and output tuples as

- a *segmentation* of structured data $(1 : m)$,
- a *block transformation* of transcoded data $(1 : 1)$, and
- a *concatenation* of fragmented data into a composite $(n : 1)$.

Arbitrary $n : m$ transformations can be composed from segmentations, block transformations and concatenations. A transformation may optionally have additional parameters that control the bijective mapping.

$$t : \mathbb{B}^n \leftrightarrow \mathbb{B}^m, n \geq 1, m \geq 1 \tag{7}$$

The bijective mapping between an internal representation m_γ and an external representation m_d as defined through transformations gives rise to a *data format instance*.

DEFINITION 3.12 (DATA FORMAT INSTANCE): Given a pair of representations (m_d, m_γ) with $m_d = \{i_0\}, m_\gamma = \{i_1, \ldots, i_n\}, n \geq 1$, a *data format instance* is a rooted, directed, acyclic graph as a *causality graph* on labeled bit sequences. The graph has root in i_0 and has $i_1, \ldots, i_n$ as its leaves. It is composed from a finite set of transformations, where each transformation t defines directed arcs from its input to its output bit sequences. Regarding intermittent nodes in the causality network, their bit sequences is the result of transformations, while their labels functionally depend on neighbouring labeled bit sequences as well as on optional context information depending on d.

As we now have defined the concept of a data format instance, we can proceed towards data formats as potentially infinite sets of data format instances.

3.2.5. Mapping between sets of representations

DEFINITION 3.13 (INTERNAL VALIDATION FUNCTION): For a given sender α and data format d, an *internal validation function* is defined as v_d^α (Eq. 8). An internal representation m_α is *valid* iff $v_d^\alpha(m_\alpha) = 1$. The subset of all valid internal representations of α for d is defined as $\mathbb{IR}_d^\alpha \subseteq \mathbb{IR}$.

$$v_d^\alpha : \mathbb{IR} \rightarrow \{0, 1\} \tag{8}$$

As a data format typically does not provide means for transporting arbitrarily labeled bit sequences, a sender α tests whether an internal representation is valid or not prior to creating a corresponding external representation.

DEFINITION 3.14 (EXTERNAL VALIDATION FUNCTION): For a given $d \in \mathbb{D}$, we define an *external validation function* v_d (Eq. 9). An external representation m_d is *valid* iff $v_d(m_d) = 1$. The subset of all valid external representations for d is defined as $\mathbb{ER}_d \subseteq \mathbb{ER}$.

$$v_d : \mathbb{ER} \rightarrow \{0, 1\} \tag{9}$$

A received external representation m'_d may be invalid, for example due to a degrading storage medium or due to interference on a network link. A receiver β thus must test whether the received m'_d is valid.

In order to transport information from the internal representation $m_\alpha \in \mathbb{IR}_d^\alpha$ to the external representation $m_d \in \mathbb{ER}_d$, and vice versa from a valid external representation $m'_d \in \mathbb{ER}_d$ to the internal representation $m'_\beta \in \mathbb{IR}_d^\beta$, a suited mapping between both sets becomes necessary.

DEFINITION 3.15 (MAPPING FUNCTION): For a given sender α and data format $d \in \mathbb{D}$, a bijective *mapping function* $f_{\alpha \rightarrow d}$ (Eq. 10) maps from $\mathbb{IR}_d^\alpha$ to $\mathbb{ER}_d$ through *encoding* and *serialization*. For a given receiver β and data format $d \in \mathbb{D}$, its inverse $f_{\beta \rightarrow d}^{-1}$ (Eq. 11) maps from $\mathbb{ER}_d$ to $\mathbb{IR}_d^\beta$ through *parsing* and *decoding*.

$$f_{\alpha \rightarrow d} : \mathbb{IR}_d^\alpha \rightarrow \mathbb{ER}_d \tag{10}$$

$$f_{\beta \rightarrow d}^{-1} : \mathbb{ER}_d \rightarrow \mathbb{IR}_d^\beta \tag{11}$$

Both sets $\mathbb{ER}_d$ and $\mathbb{IR}_d^\alpha$ have the same size due to the bijectivity of export and import functions. Both sets $\mathbb{ER}$ and $\mathbb{IR}$ are infinite. Depending on the data format d, the sets $\mathbb{ER}_d$ and $\mathbb{IR}_d^\alpha$ may be finite. Building on the previous definitions, the notion of a *data format* can now be defined.

DEFINITION 3.16 (DATA FORMAT): A data format d is a possibly infinite set of data format instances, which map between a normative $\mathbb{ER}_d \subseteq \mathbb{ER}$ and a canonical $\mathbb{IR}_d^\gamma \subseteq \mathbb{IR}$ and which makes assumptions on the underlying channel c.

3.3. Elemental properties

We can observe elemental properties of a data format for sender and receiver along the flow of information in Figure 1:

- *The sender α can decide whether the internal representation m_α is valid using function v_d^α.* For example, given GIF89a as data format and m_α as an image, the

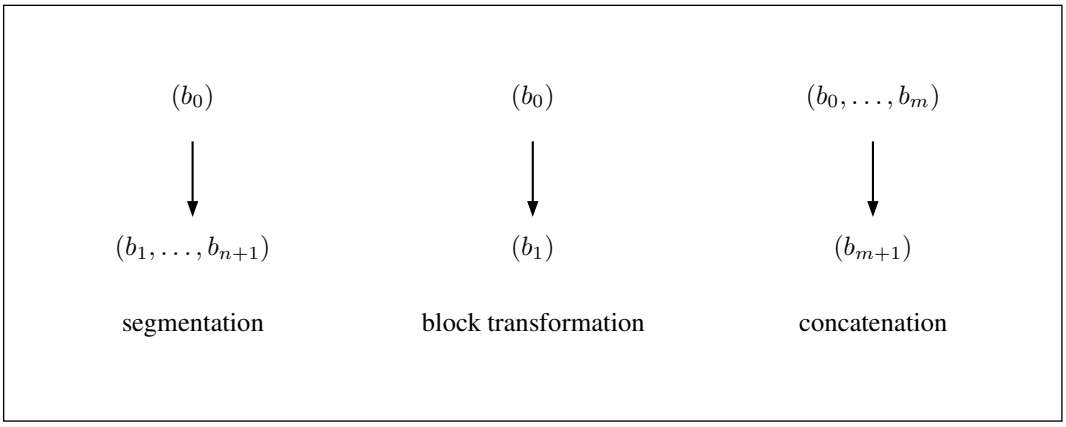

Fig. 2. Transformations ordered by input and output cardinality.

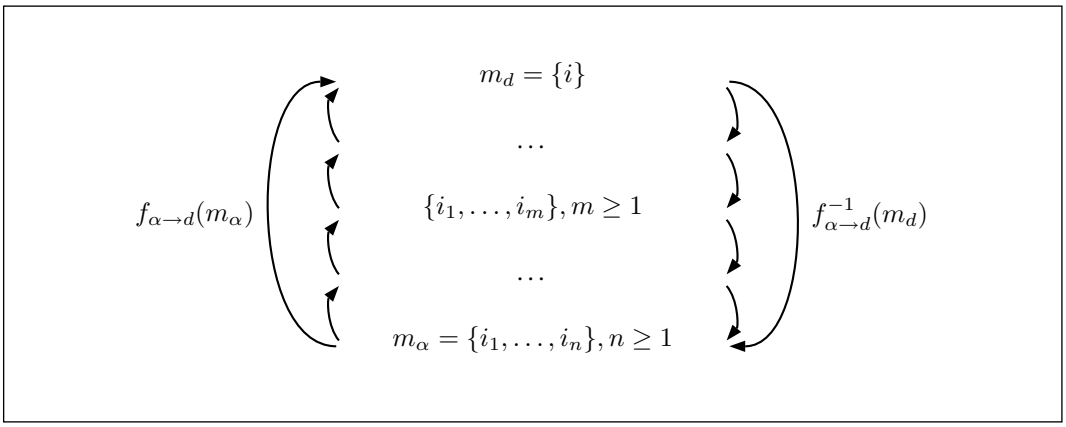

Fig. 3. Bijective mapping between m_α and m_d

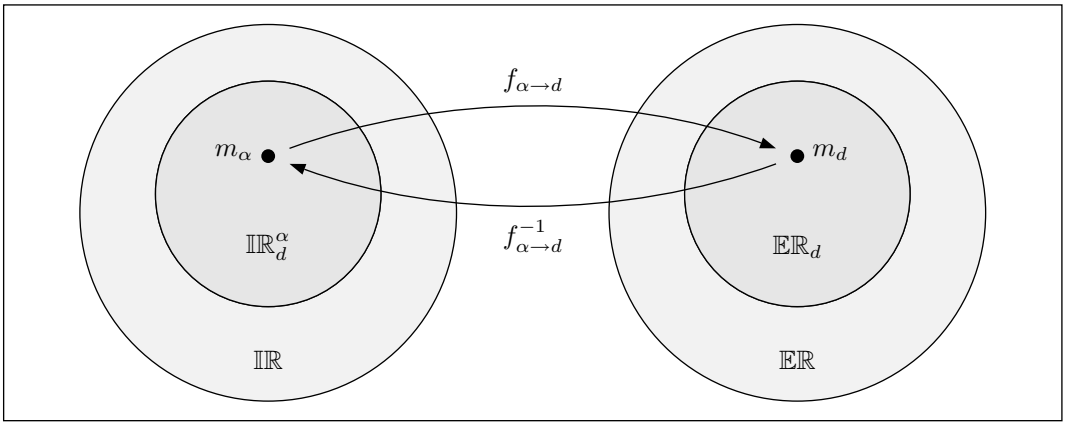

Fig. 4. Bijective mapping between internal representations $\mathbb{IR}^\alpha_d$ and external representations $\mathbb{ER}_d$ through mapping function $f_{\alpha \to d}$ and its inverse $f^{-1}_{\alpha \to d}$

validation function v_d^α would fail if the width of the image were 65536 pixels, as the data format constraints it to $2^{16} - 1 = 65535$ pixels or less.

- *The sender α can compute a valid external representation m_d from a valid internal representation m_α using the bijective function $f_{\alpha \to d}$.* This computation includes steps such as encoding values and serializing the external representation.
- *The receiver β can decide whether the external representation m'_d is valid using function v_d.* The channel c may have introduced errors that invalidate the external representation m'_d, such as bit flips during storage on a deteriorating medium, or interference on a network link during transmission.
- *The receiver β can compute a valid internal representation m'_β from a valid external representation m'_d using the bijective function $f^{-1}_{\beta \to d}$.* This mapping includes steps such as parsing the external representation and decoding its values.

For a formalization of data formats, it thus is desirable for a model on data formats to guarantee *bijectivity*, *decidability* as well as its overall *consistency*.

3.4. Limits for modelling arbitrary data formats

An ideal model would guarantee both the decidability of functions v_d^α, v_d, $f_{\alpha \to d}$ and $f^{-1}_{\beta \to d}$ as well as the bijectivity of functions $f_{\alpha \to d}$ and $f^{-1}_{\beta \to d}$. Moreover, it would guarantee the consistency of validation and mapping functions. Based on established results from formal languages and computational theory, we will now show to which degree this is possible.

3.4.0.1. Decidability

To guarantee decidability, we ideally would like to model the set of *deciders*, that is, machines that always halt. Yet, the problem of deciding whether an Turing Machine (TM) accepts a given input is the well-known *Halting Problem* A_{TM} (Eq. 12), which is undecidable [32, 33].

$$A_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and accepts } w\} \tag{12}$$

Reducing the computational power of a TM to a linear-bounded automaton (LBA), decidability can be guaranteed again, as the corresponding problem A_{LBA} is decidable [33]. Yet, a reduction from TMs to LBAs excludes valid deciders from being modelled as well.

3.4.0.2. Bijectivity

To guarantee bijectivity, we can represent a mapping function as a *reversible automaton*, where reversibility is ensured through its bijective transition function σ. Due to a finite set of states and an alphabet of finite size, the transition function consists of a finite number of transition formulas, which can be represented as tuples and for which bijectivity can be decided. Bennett proved the Reversible Turing Machine (RTM) as equivalent to the TM in computational power by showing that a three-tape RTM can compute a single-tape TM [34].

3.4.0.3. Consistency

Consistency between bijective mapping functions $f_{\alpha \to d}$ and its inverse $f^{-1}_{\alpha \to d}$ can be guaranteed through a reversible automaton. As well, consistency between $f_{\alpha \to d}$ and v_d^α for sending as well as $f^{-1}_{\alpha \to d}$ and v_d for receiving can be guaranteed for decidable bijective mapping functions by defining v_d^α and v_d to return 1 iff the respective $f_{\alpha \to d}$ and $f^{-1}_{\alpha \to d}$ accepts and thus decides the input, and 0 otherwise.

3.4.0.4. Summary

Computational theory places significant limits on modelling arbitrary data formats in general. A model can guarantee bijectivity, yet no model can both cover arbitrary data formats and still guarantee decidability. We thus opt for a second-best approach by dropping guaranteed decidability as property in favor of a generic approach.

4. MODELS

4.1. Data Format Instances

The following definition of the *Bitstream Segment Graph* from the original publications in [1, 6] has been adapted to the context of this article and extended with a list of operations and an RDF/N3-based storage representation.

4.1.1. Definition

For defining a model to describe the composition of data in a canonic form, we now define the *Bitstream Segment Graph* building on the causality network idea from the analysis.

DEFINITION 4.1 (BITSTREAM SEGMENT): Given a bitstream segment $v \in V$, the set of bitstream segments V, the set of finite consecutive bit sequences $B = \{0,1\}^n, n \in \mathbb{N} \setminus \{0\}$ and $\varphi : V \mapsto B$, then the bitstream segment v represents a finite consecutive bit sequence $\varphi(v) \in B$.

DEFINITION 4.2 (BITSTREAM SOURCE): A bitstream source is a root bitstream segment $v_{Root} \in V$ with a defined $\varphi(v_{Root})$.

A bitstream source represents a digital item which is composed according to a data format. Files, network packets or file systems on some storage medium are examples for octet-aligned bitstream sources.

DEFINITION 4.3 (BITSTREAM ENCODING): Given a bitstream encoding $e = (rel, v, l) \in R_E, v \in V, l \in L$, where R_E is the set of bitstream encodings and L is the set of literals,

Used in encoding $e \in R_E$?	Used in transformation $t \in R_T$?		Type	RDF Type
no	no	(as input)	Generic	`bsg:generic`
yes	no	(as input)	Primitive	`bsg:primitive`
no	segmentation	(as input)	Structure	`bsg:structure`
no	transformation	(as input)	Transcode	`bsg:transcode`
no	concatenation	(as input)	Fragment	`bsg:fragment`
no	concatenation	(as output)	Composite	`bsg:composite`

Table 2. Types of bitstream segments.

then for a given v, e specifies a mapping relation $rel(\varphi(v), l)$, required to be bijective. It is abbreviated with $\phi(v) = l$, where $\phi : V \mapsto L$.

A bitstream segment can represent an encoded literal that is part of the data contained in a bitstream source (a primitive). For example, there are two bitstream segments within a PNG file which contain encoded integers that represent the width and height of the image.

DEFINITION 4.4 (BITSTREAM TRANSFORMATION): Given a bitstream transformation $t = (rel, V_{in}, V_{out}, P) \in R_T$, where V_{in}, V_{out} are totally ordered sets with $V_{in} \subset V, V_{out} \subset V, V_{in} \neq \emptyset, V_{out} \neq \emptyset, V_{in} \cap V_{out} = \emptyset$, R_T is the set of bitstream transformations and P is the set of parameters, then t specifies a mapping relation $rel(V_{in}, V_{out}, P)$, required to be bijective, between V_{in} and V_{out} under application of P.

In general, a bitstream transformation t bijectively maps a set of input bitstream segments V_{in} as *predecessors* to a a set of new bitstream segments V_{out} as *successors* as result of the transformation. *Normalized bitstream transformations* categorized by $|V_{in}| : |V_{out}|$ cardinality are

- the concatenating transformation of multiple fragment segments into one composite segment $(m : 1)$ (for fragments),
- a class of block transformations such as decompression or decryption $(1 : 1)$ (for transcodes) and
- segmenting transformation of a structured segment into multiple separate bitstream segments $(1 : n)$ (for structures).

Arbitrary transformations of $m : n$ cardinality can be composed by concatenating two or more normalized transformations.

DEFINITION 4.5 (BITSTREAM SEGMENT GRAPH): Given a set of bitstream transformations R_T and a set of bitstream encodings R_E, then R_T and R_E induce a bitstream segment graph (BSG). It is a weakly connected, directed acyclic rooted graph $G = (V, E)$ with a set of bitstream segments V as vertices and a set of directed edges $E \subset V \times V$, connecting transformation input/output pairs of bitstream segments. A BSG describes the composition of a bitstream source and is complete iff

$$\forall v \in V\colon (\exists! t = (rel_t, V_{in}, V_{out}, P) \in R_T, v \in V_{in}) \oplus (\exists! e = (rel_e, v_e, l) \in R_E, v = v_e)$$

A BSG is composed from bitstream transformations and encodings, which are required to have bijective mapping relations. It therefore provides a bijective mapping between its bitstream source and its contained literals.

4.1.1.1. Types of Bitstream Segments

In a BSG instance, bitstream segments are categorized into one of 6 *types* as *generic*, *primitive*, *structure*, *transcode*, *fragment* and *composite*, depending on their participation in normalized bitstream transformations and bitstream encodings as shown in Table 2. While the concept of primitives, structures, transcodes and fragments have been previously defined in Section 2, a generic serves for data of yet unknown type in an incomplete description, and a composite describes the aggregation of two or more fragments.

To prevent a conflicting type assignment for bitstream segments that have both the "upward" type and another "downward" type such as a composite that contains a structure, an identity transformation is inserted after the composite and the "downward" type such as the structure is assigned to the newly inserted bitstream segment.

4.1.1.2. Coverage of Bitstream Segments

The *coverage* of a bitstream segment is a measure in the range between 0 and 1 and expresses how completely a bitstream segment is mapped to encoded literals through its successor(s). It is defined as 1 for primitives, 0 for generics, and computed as length-weighted sum over the coverage of all successors otherwise. For example, for a structure bitstream segment `a` with two primitive segments as successors, the coverage of `a` would be 1. In case of one primitive segment and a generic segment of equal length as successors, the coverage of `a` would be 0.5. The coverage of a BSG instance refers to that of its bitstream source.

4.1.2. Composition Algorithm

Using definitions 4.1 to 4.5, we are able to describe the bijective mapping between a bitstream source and its set of contained literals. The following simple algorithm constructs a BSG step-by-step. For a construction at step x, the tuple

$$(v_{Root}, V_x, V_{leaf_x}, V_{literal_x}, R_{T_x}, R_{E_x})$$

describes a designated root bitstream segment v_{Root}, a set of bitstream segments V_x, a set of leaf bitstream segments V_{leaf_x}, a set of literal bitstream segments $V_{literal_x}$, a set of bitstream transformations R_{T_x} and a set of bitstream encodings R_{E_x}, whereas initial values are

$$\begin{aligned} V_0 &= \{v_{Root}\} \\ V_{leaf_0} &= \{v_{Root}\} \\ V_{literal_0} &= \emptyset \\ R_{T_0} &= \emptyset \\ R_{E_0} &= \emptyset \end{aligned}$$

Starting at step $x = 1$, each step either adds a transformation or an encoding through an operation. For a transformation, the addition of $t = (rel, V_{in}, V_{out}, P) \notin R_{T_{x-1}}$, $V_{in} \subseteq V_{leaf_{x-1}}$ results in

$$\begin{aligned} V_x &= V_{x-1} \cup V_{out} \\ V_{leaf_x} &= V_{leaf_{x-1}} \cup V_{out} \setminus V_{in} \\ V_{literal_x} &= V_{literal_{x-1}} \\ R_{T_x} &= R_{T_{x-1}} \cup \{t\} \\ R_{E_x} &= R_{E_{x-1}} \end{aligned}$$

whereas the addition of an encoding $e = (rel, v, l) \notin R_{E_{x-1}}, v \in V_{leaf_{x-1}}$ results in

$$\begin{aligned} V_x &= V_{x-1} \\ V_{leaf_x} &= V_{leaf_{x-1}} \setminus v \\ V_{literal_x} &= V_{literal_{x-1}} \cup \{l\} \\ R_{T_x} &= R_{T_{x-1}} \\ R_{E_x} &= R_{E_{x-1}} \cup \{e\} \end{aligned}$$

For step y, the tuple induces a BSG $G_y = (V_y, E_y)$ where E_y is defined as follows:

$$\begin{aligned} &\forall t = (rel, V_{in}, V_{out}, P) \in R_{T_y}, \\ &\forall v_s \in V_{in}, \forall v_t \in V_{out} : e = (v_s, v_t) \in E_y \end{aligned}$$

For a complete BSG, these steps are repeated until $V_{leaf_z} = \emptyset$, where the algorithm terminates as no further addition of either transformation or encoding to leaf bitstream segments is possible.

4.1.3. Operations

For manual annotation of a bitstream source, it is helpful to break up transformations into a number of smaller, incremental operations. For that purpose, we define a set of parameterized operations under which a BSG instance is closed. Listing these with their respective inverse in pairs, these are as follows:

- **initial_split, final_join**: Replaces a generic segment a with a structure segment b, splits a into two consecutive generic segments a_1, a_2 and adds both in that order as successors to the structure b.
- **split, join**: Under a structure a, replaces a generic segment b with two consecutive generic segments b_1, b_2 in that order which result from splitting b in two.
- **tie, untie**: Replaces a consecutive set of segments A with a structure segment b that has A as its successors.
- **declare_primitive, undeclare_primitive**: Transforms a generic segment a into a primitive segment a.
- **expand, compress**: Transforms a generic segment a into a transcode segment a and adds a generic segment b as its successor.
- **declare_fragment, undeclare_fragment**: Transforms a generic segment a into a fragment segment a.
- **compose, decompose**: Aggregates an ordered set of fragments A into a composite segment b, assigns b as sole successor to each fragment $a \in A$, and adds a generic segment c as sole successor of b.

4.1.4. Storage Representation

For exchanging a BSG instance which describes the composition of binary data, we can now define an RDF vocabulary for the Bitstream Segment Graph model. For expressing a BSG instance using RDF, bitstream segments are represented as RDF resources, belonging to certain classes and having certain properties. For storing BSG instances in RDF, Notation 3 is used [35]. In the following definitions and examples, namespaces and prefixes are used according to Table 3.

4.1.4.1. RDF Classes

Every bitstream segment has a `rdf:type` value of both `bsg:segment` and the specific RDF class corresponding to its type as listed from Table 4, such as `bsg:primitive`. A root bitstream segment additionally has a `rdf:type` of `bsg:source`. It is worth noting that the normalized bitstream transformations of segmentation, block transformation and concatenation from Definition 4.4 correspond to the classes `bsg:structure`, `bsg:transcode` and `bsg:composite`, respectively.

Prefix	Namespace	Comment
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#	Standard RDF namespace
bsg	http://www.dataformats.net/2009/01/25-bsg-syntax-ns#	BSG namespace
bsge	http://www.dataformats.net/2009/01/25-bsg-ext-ns#	BSG Extension namespace

Table 3. RDF namespace declarations

RDF Class	Description
bsg:source	Class for bitstream sources
bsg:segment	Abstract base class for bitstream segments
bsg:generic	Class for bitstream segments where the purpose is undefined
bsg:primitive	Class for bitstream segments representing an encoded literal
bsg:structure	Class for bitstream segments composed from two or more bitstream segments with separate, distinct meaning
bsg:transcode	Class for bitstream segments representing a transcoded bit sequence
bsg:fragment	Class for bitstream segments representing a fragment of a larger bit sequence with a uniform meaning
bsg:composite	Class for bitstream segments representing a bit sequence with a uniform meaning aggregated from two or more fragments

Table 4. RDF classes for bitstream segments.

RDF Class	RDF Property	Cardinality	Description
bsg:source	bsg:href	1..1	Reference to a bitstream source
bsg:segment	bsg:start	1..1	Start position in bits (inclusive)
	bsg:length	1..1	Length in bits
	bsg:end	1..1	End position in bits (exclusive)
	bsg:semantics	0..n (size of list)	Identifier for format-specific semantics
	bsg:predecessor	0..n (size of list)	Ordered list of predecessors (input)
	bsg:successor	0..n (size of list)	Ordered list of successors (output)
bsg:generic	bsg:predecessor	*0..1* (size of list)	*Restriction: Generics have at most one predecessor*
	bsg:successor	*0..0*	*Restriction: Generics do not have successors*
bsg:primitive	bsg:encoding	1..1	Identifier for the encoding used
	bsg:predecessor	*0..1* (size of list)	*Restriction: Primitives have at most one predecessor*
	bsg:successor	*0..0*	*Restriction: Primitives do not have successors*
bsg:structure	bsg:predecessor	*0..1* (size of list)	*Restriction: Structures have at most one predecessor*
	bsg:successor	*2..n* (size of list)	*Restriction: Structures have at least two successors*
bsg:transcode	bsg:codec	1..1	Identifier for the codec used
	bsg:predecessor	*0..1* (size of list)	*Restriction: Transcodes have at most one predecessor*
	bsg:successor	*1..1* (size of list)	*Restriction: Transcodes have exactly one successor*
bsg:fragment	bsg:predecessor	*1..1* (size of list)	*Restriction: Fragments have exactly one predecessor*
	bsg:successor	*1..1*	*Restriction: Fragments have exactly one successor*
bsg:composite	bsg:predecessor	*2..n* (size of list)	*Restriction: Composites have at least two predecessors*
	bsg:successor	*1..1* (size of list)	*Restriction: Composites have exactly one successor*

Table 5. RDF properties for bitstream segments.

4.1.4.2. RDF Properties

Depending on the RDF class, bitstream segments have specific properties according to Table 5. For placement, every bitstream segment has a `bsg:start`, `bsg:length` and `bsg:end` property with integer values. These refer to its exact placement within the bit sequence composed from its predecessor(s), or within its defined bit sequence in case of a bitstream source. A root bitstream segment always starts at 0. All three properties are measured in bits, whereas the start position is included and the end position excluded, which simplifies testing two bitstream segments for neighbourship.

Regarding their composition, every bitstream segment besides the bitstream source has a `bsg:predecessor` property refering to a nonempty RDF list of bitstream segment URIs. Likewise, every bitstream segment besides `bsg:generic` or `bsg:primitive` segments have a `bsg:successor` property referring to a nonempty RDF list of bitstream segment URIs. Class-specific restrictions listed in Table 5 apply which correspond to the underlying BSG model. Only the bitstream source has the `bsg:source` property.

The meaning of a bitstream segment can be assigned a `bsg:semantics` property referring to a nonempty RDF list of string literals. For example, this could refer to *PNG Signature* semantics using `png:signature` as value. For `bsg:primitive` and `bsg:transcode` bitstream segments, the identification of the actual encoding or codec used is given through the `bsg:encoding` and `bsg:codec` properties, respectively. For example, this could include an unsigned integer encoding (most significant bit first), referred to by `bsge:encoder-msbfuint` or a *gzip* transformation as `bsge:transcoder-gzip`. The normative definition of concrete identifiers for semantics, encodings and codecs depends on the data format to be described and is a standardization effort which is not within the scope of this publication.

4.1.5. Visual Representation

Depending on the type, segments in a BSG are depicted as shown in Figure 6, where *start* and *end* denote inclusive start and exclusive end bit positions relative to the parent bitstream segment(s), *type* denotes the bitstream segment type, *parameter* denotes a parameter for some types and *id* denotes some plaintext identification.

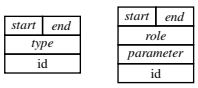

Fig. 6. Visual representations; generic, structure and composite bitstream segments (left); fragment, primitive and transcode bitstream segments (right)

4.1.6. Tool Support

We have developed the *Apeiron BSG Editor* as a tool for the annotation of bitstreams using the BSG model through a combination of graph visualization and table representation of binary data, where operations can be applied. It is written in Java, is based on the *Open Services Gateway initiative (OSGi) R4* specification and uses the *Prefuse Visualization Toolkit* [36]. Apeiron is available online and can be launched from `http://www.dataformats.net` via Java WebStart.

4.2. Data Formats

The following definition of *BSG Reasoning* originally published in [7] has been adapted to the context of this article.

4.2.1. Definition

For defining a data format, we need to define a (potentially infinite) set of data format instances which we can represent using the BSG approach. We define such a set through the *BSG Reasoning* approach as the set of stable models resulting from first-order logic rules on the BSG model, expressed as implications or biconditionals. For rules, predicates are used that refer to either deduced or computed facts. In terms of existing logic languages, the BSG Reasoning approach resembles Datalog [37] extended with functions.

4.2.1.1. Facts

A fact is represented as a predicate where all parameters are ground. For example, the start position of a bitstream segment $a1$ at bit 0 is represented as `bsg:start(a1,0)`.

4.2.1.2. Predicates

Deducible predicates refer to facts that were either given initially or subsequently deduced through rules. They are not limited to BSG-related properties and relations only, but may also include predicates for intermittent facts which may be needed for deducing a BSG instance. For deduced predicates, the open world assumption applies, as a currently unknown fact may become known later. *Computable predicates* refer to facts that can be computed directly, listed in Table 6. They handle aspects such as decoding the literal `?l` of a primitive bitstream segment `?x` from the so-far deduced, partial BSG instance through `bsg:value(?x,?l)`, or for solving the equation ?v=?u+1 through `math:sum(?u,1,?v)` if either `?u` or `?v` are known. These predicates can choose between the open world assumption and the closed world assumption, as they can decide to refute facts that will always fail, such as `math:sum(1,2,4)`.

Predicates have parameters that can either be *ground* and thus have a specific value, or be a *variable*. A *mode* of a predicate states for each of its parameters whether it is ground or variable. Computable predicate may support arbitrary modes, eg. allowing `math:sum` the computation of

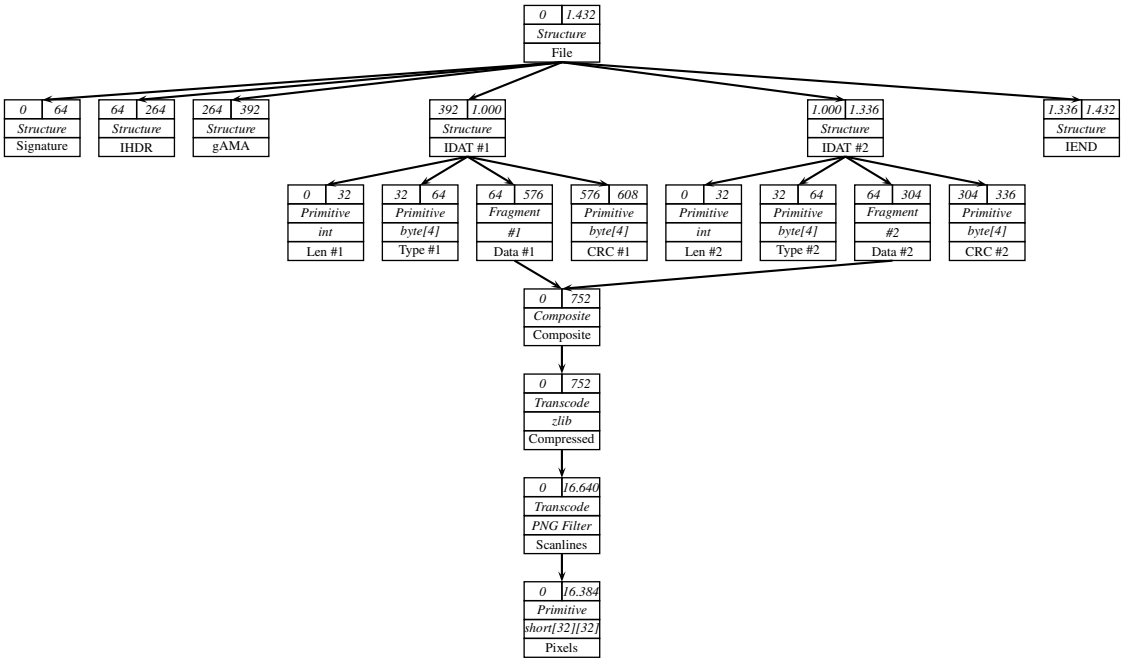

Fig. 5. Partial bitstream segment graph for file "oi2n0g16.png", showing the bijective mapping of two PNG IDAT chunks to a 16 bit grayscale image with a resolution of 32×32 pixel.

`math:sum(?u,4,5)` as well as `math:sum(1,?v,5)` and `math:sum(1,4,?w)`.

4.2.1.3. Rules

Using these types of predicates, we can build rules as implications or biconditionals. These rules can be partitioned into model-specific rules that capture properties and relations inherited from the BSG model itself, and format-specific rules that represent data format knowledge. For example, a BSG-specific rule is that two neighbouring bitstream segments `b` and `c` share a boundary, so from the facts `bsg:follows(b,c)` and `bsg:end(b,512)`, the fact `bsg:start(c,512)` follows. In Figure 5, a format-specific rule of PNG is that if a segment a represents a PNG file as stated by the fact `bsg:semantics(a,png:file)`, then its first successor b is a PNG signature, which is expressed by the facts `bsg:firstSuccessor(a,b)` and `bsg:semantics(b,png:signature)`.

4.2.1.4. Reasoning process

For deducing a BSG instance, initial knowledge on a bitstream source is given, such as the fact `bsg:source(a,'oi2n0g16.png')`. Through a series of iterative steps, the set of rules is applied in a monotone deduction process. In each step for every rule, it is tried to match the antecedents of a rule previously deduced knowledge. If the antecedents of a rule matches, then for its conclusion, the computable predicates are tested and the deducible predicates are deduced. Should a computable predicate fail in this test, the reasoning process aborts, as a conclusion does not hold. This allows the use of validation rules that assert certain properties, eg. that for all bitstream segments, its respective `bsg:start` and `bsg:length` have to sum up to its `bsg:end`, which can be violated in case of contradictory information contained in a damaged or maliciously crafted bitstream source. When no new facts are deduced in a step, then a fixed point consisting of the deducible facts of a BSG instance is reached.

If a fixed point is reached, the resulting BSG facts can then be translated into a BSG instance for that bitstream source. This requires post-processing steps such as assigning the generic bitstream segment type whenever no type was deduced for a bitstream segment. The deduction of a BSG instance therefore can either

- abort with a computable predicate refuting a fact in a rule conclusion, indicating that a conclusion does not hold and thus the bitstream source does not conform to the specified data format,
- reach a fixed point with a coverage $x < 1$, indicating that there are bitstream segments in this data format instance not specified in the set of rules, or
- reach a fixed point with a coverage $x = 1$, indicating that this data format instance is completely covered by the set of rules.

Building a set of rules as data format knowledge is typically an incremental process. It starts with the collection of bitstream sources for a corpus that represents a specific format, and the definition of an initial set of rules. This set of rules can be improved step-by-step by computing the BSG instance for every bitstream source in the corpus and computing its coverage. One then can select BSG instances with a coverage $x < 1$ and focus on generic bitstream segments which need to be described further through additional rules. Actual knowledge on how these generic bitstream segments are actually composed may come from consulting textual specifications, existing implementations or through try-and-error reverse engineering efforts. Repeating this process increases the overall coverage of BSG instances in the corpus. For a corpus, an fitting set of rules is found if the coverage reaches 1 for all of its BSG instances.

4.2.2. Implementations

In order to test the BSG Reasoning approach in practice, we have developed a suited fixed-point reasoning system in Java for BSG reasoning. We thereby defined suited interfaces for processing bitstream transformations and bitstream encodings, and implemented components for handling transformations and encodings as required for the PNG image file format.

5. EXAMPLES

5.1. Describing the composition of data automatically

In order to demonstrate the BSG Reasoning approach, we describe a small subset of the Portable Network Graphics (PNG) image format. We required that of this subset, some data format instances should at least be sufficiently complex as to require all four types of descriptive capabilities (structures, primitives, transcodes and fragments) including functional dependencies as provided by the BSG model.

5.1.1. Setup

We identified a suited subset of PNG images, namely those where compressed image data is stored as separate fragments in so-called *IDAT chunks*. For building a suited corpus, we examined the PNG Test Suite [38] with 156 PNG images for compliance testing, including corrupted files and extreme variants, and selected 8 images with filename pattern `oi??????.png`.

Regarding the granularity of description, we allowed primitive bitstream segments to represent arrays of encoded literals. Without this consideration, the decomposition of arrays such as pixel data into individual pixels would have bloated the resulting description of a data format instance without substantial benefit.

5.1.2. Data format rules

We built a fitting set of rules for our corpus, consisting of 17 model-specific rules (see Table 7) and 36 format-specific rules (see Table 8 for an excerpt).

Regarding model-specific rules, we start with rules on placement regarding a bitstream segment. This begins with a rule for deducing `bsg:start` and `bsg:length` from an initially given `bsg:source` (M1). If any two of `bsg:start`, `bsg:length` and `bsg:end` are given for a bitstream segment, the remaining fact can be deduced (M2-M4). Moreover, if all facts are given for a bitstream segment, it can be validated for ensuring consistency (M5). Further rules include aspects of neighbourship of bitstream segments in a structure (M6 & M7), successorship of bitstream segments (M8-M12), placement in a structure (M13-M15) and resolvability (M16 & M17), which is necessary for decoding the contained literal of primitive bitstream segments.

Finally, we come to format-specific rules on our PNG subset. We start with a rule that deduces the PNG-specific type of 'png:root' for a bitstream source (F1). For such a bitstream segment, we can deduce that there exists a first successor `?s` with `bsg:semantics(?s,'png:signature')` (F2). For a 'png:signature', there exists a following 'png:chunk' structure (F3) as shown in Figure 7, which again always begins with a 'png:chunk-length' bitstream segment (F4), followed by a 'png:chunk-type' bitstream segment (F5).

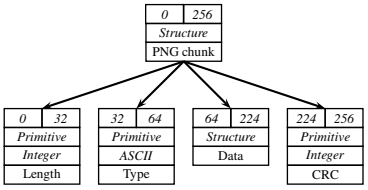

Fig. 7. BSG instance for a PNG chunk.

If the value of a 'png:chunk-length' is 0, then the 'png:chunk-type' is followed directly by the 'png:chunk-crc' bitstream segment as last successor of the chunk (F6). Otherwise, the 'png:chunk-type' bitstream segment is followed by a variable-length 'png:chunk-data' bitstream segment and again the 'png:chunk-crc' bitstream segment (F7). Details on bitstream segments such as their type, encoding and length are provided for 'png:signature' (F8), 'png:chunk-length' (F9), 'png:chunk-type' (F10) and 'png:chunk-crc' (F11) bitstream segments. The PNG-specific type of the chunk is deduced from the 'png:chunk-type' value and assigned as `bsg:semantics` to the chunk (F12). The remaining rules listed in Table 8 state that if there is space left after a chunk, there exists another one following (F13), otherwise the chunk is the last successor of the bitstream source (F14). Further rules handle chunk-specific aspects, eg. for the IHDR chunk

Predicate	Behaviour
`math:lt(?a,?b)`	Tests the formula `?a` $<$ `?b`.
`math:lte(?a,?b)`	Tests the formula `?a` $\leq$ `?b`.
`math:product(?a,?b,?c)`	Computes the formula `?a` $\cdot$ `?b` $=$ `?c` if two parameters are ground and no division by zero occurs, and assigns the result to the third variable parameter. Tests the formula if all parameters are ground.
`math:sum(?a,?b,?c)`	Computes the formula `?a` $+$ `?b` $=$ `?c` if two parameters are ground and assigns the result to the third variable parameter. Tests the formula if all parameters are ground.
`util:concat(?a,?b,?c)`	Concatenates ground strings `?a` and `?b` and binds the result to variable `?c`. Tests whether the concatenation of `?a` and `?b` corresponds to `?c` if all parameters are ground.
`util:sourceLength(?a,?b)`	Gets the length in bits of the ground file `?a` and binds it to variable `?b`. Tests whether file `?a` has length `?b` in bits if both are ground.
`util:skolem(?a,...,?c)`	*Skolem function providing for existential quantification*. Maps the set of ground parameters (`?a`, ...) to a value and binds it to variable `?c`. Maps a ground `?c` to a set of values and binds them to variables (`?a`, ...). Tests whether (`?a`, ...) and `?c` map to each other if all parameters are ground.
`util:value(?a,?b)`	Decodes the contained literal of a ground primitive bitstream segment `?a` if it is `bsg:resolved`, and assigns the result to variable `?b`. Tests whether the bitstream segment `?a` contains the literal `?b` if both parameters are ground.

Table 6. List of computable predicates.

#	Rule
M1	`bsg:source(?a,?f)` $\wedge$ `util:sourceLength(?f,?l)` $\rightarrow$ `bsg:start(?a,0)` $\wedge$ `bsg:length(?a,?l)`
M2	`bsg:length(?a,?l)` $\wedge$ `bsg:end(?a,?e)` $\wedge$ `math:sum(?s,?l,?e)` $\rightarrow$ `bsg:start(?a,?s)`
M3	`bsg:start(?a,?s)` $\wedge$ `bsg:end(?a,?e)` $\wedge$ `math:sum(?s,?l,?e)` $\rightarrow$ `bsg:length(?a,?l)`
M4	`bsg:start(?a,?s)` $\wedge$ `bsg:length(?a,?l)` $\wedge$ `math:sum(?s,?l,?e)` $\rightarrow$ `bsg:end(?a,?e)`
M5	`bsg:start(?a,?s)` $\wedge$ `bsg:length(?a,?l)` $\wedge$ `bsg:end(?a,?e)` $\rightarrow$ `math:sum(?s,?l,?e)`
M6	`bsg:leads(?a,?b)` $\leftrightarrow$ `bsg:follows(?b,?a)`
M7	`bsg:leads(?a,?b)` $\wedge$ `bsg:end(?a,?p)` $\leftrightarrow$ `bsg:follows(?b,?a)` $\wedge$ `bsg:start(?b,?p)`
M8	`bsg:firstSuccessor(?a,?b)` $\rightarrow$ `bsg:successor(?a,?b)`
M9	`bsg:lastSuccessor(?a,?b)` $\rightarrow$ `bsg:successor(?a,?b)`
M10	`bsg:successor(?a,?b)` $\rightarrow$ `bsg:predecessor(?b,?a)`
M11	`bsg:successor(?a,?b)` $\wedge$ `bsg:leads(?b,?c)` $\rightarrow$ `bsg:successor(?a,?c)`
M12	`bsg:successor(?a,?b)` $\wedge$ `bsg:follows(?b,?c)` $\rightarrow$ `bsg:successor(?a,?c)`
M13	`bsg:firstSuccessor(?a,?b)` $\rightarrow$ `bsg:start(?b,0)`
M14	`bsg:lastSuccessor(?a,?b)` $\wedge$ `bsg:length(?a,?c)` $\rightarrow$ `bsg:end(?b,?c)`
M15	`bsg:lastSuccessor(?a,?b)` $\wedge$ `bsg:end(?b,?c)` $\rightarrow$ `bsg:length(?a,?c)`
M16	`bsg:start(?a,?s)` $\wedge$ `bsg:length(?a,?l)` $\wedge$ `bsg:end(?a,?e)` $\wedge$ `bsg:type(?a,?t)` $\wedge$ `bsg:source(?a,?f)` $\rightarrow$ `bsg:resolved(?a)`
M17	`bsg:successor(?a,?b)` $\wedge$ `bsg:start(?b,?s)` $\wedge$ `bsg:type(?b,?t)` $\wedge$ `bsg:resolved(?a)` $\rightarrow$ `bsg:resolved(?b)`

Table 7. List of model-specific rules.

#	Rule
F1	bsg:source(?a,?f) → bsg:semantics(?a,'png:root')
F2	bsg:semantics(?r,'png:root') → util:skolem('F2',?r,?s) ∧ bsg:type(?r,'bsg:structure') ∧ bsg:firstSuccessor(?r,?s) ∧ bsg:semantics(?s,'png:signature')
F3	bsg:semantics(?s,'png:signature') → util:skolem('F3',?s,?f) ∧ bsg:leads(?s,?f) ∧ bsg:semantics(?f,'png:chunk')
F4	bsg:semantics(?c,'png:chunk') → util:skolem('F4',?c,?l) ∧ bsg:firstSuccessor(?c,?l) ∧ bsg:semantics(?l,'png:chunk-length')
F5	bsg:semantics(?l,'png:chunk-length') → util:skolem('F5',?l,?t) ∧ bsg:leads(?l,?t) ∧ bsg:semantics(?t,'png:chunk-type')
F6	bsg:semantics(?l,'png:chunk-length') ∧ bsg:value(?l,0) ∧ bsg:leads(?l,?t) ∧ bsg:successor(?ch,?l) → util:skolem('F6',?l,?t,?ch,?cr) ∧ bsg:lastSuccessor(?ch,?cr) ∧ bsg:leads(?t,?cr) ∧ bsg:semantics(?cr,'png:chunk-crc')
F7	bsg:semantics(?l,'png:chunk-length') ∧ bsg:value(?l,?v) ∧ math:lt(0,?v) ∧ bsg:leads(?l,?t) ∧ bsg:successor(?ch,?l) ∧ math:product(?v,8,?lv) → bsg:leads(?t,?d) ∧ bsg:leads(?d,?cr) ∧ bsg:lastSuccessor(?ch,?cr) ∧ bsg:length(?d,?lv) ∧ bsg:semantics(?d,'png:chunk-data') ∧ bsg:semantics(?cr,'png:chunk-crc')
F8	bsg:semantics(?t,'png:signature') → bsg:type(?t,'bsg:primitive') ∧ bsg:encoding(?t,'http://www.dataformats.net/2008/04/bsg-encodings# ascii-string') ∧ bsg:length(?t,64)
F9	bsg:semantics(?l,'png:chunk-length') → bsg:type(?l,'bsg:primitive') ∧ bsg:encoding(?t,'http://www.dataformats.net/2008/04/bsg-encodings# msbf-uint') ∧ bsg:length(?l,32)
F10	bsg:semantics(?t,'png:chunk-type') → bsg:type(?t,'bsg:primitive') ∧ bsg:encoding(?t,'http://www.dataformats.net/2008/04/bsg-encodings# ascii-string') ∧ bsg:length(?t,32)
F11	bsg:semantics(?cr,'png:chunk-crc') → bsg:type(?t,'bsg:primitive') ∧ bsg:encoding(?t,'http://www.dataformats.net/2008/04/bsg-encodings# msbf-uint') ∧ bsg:length(?cr,32)
F12	bsg:semantics(?ch,'png:chunk') ∧ bsg:semantics(?t,'png:chunk-type') ∧ bsg:successor(?ch,?t) ∧ bsg:value(?t,?v) → util:concat('png:chunk:',?v,?ct) ∧ bsg:semantics(?ch,?ct)
F13	bsg:semantics(?c, 'png:chunk') ∧ bsg:end(?c,?ce) ∧ bsg:successor(?r,?c) ∧ bsg:length(?r,?rl) ∧ math:lt(?ce,?rl) → util:skolem('F13',?c,?ce,?r,?rl,?nc) ∧ bsg:leads(?c,?nc) ∧ bsg:semantics(?nc,'png:chunk')
F14	bsg:semantics(?c, 'png:chunk') ∧ bsg:end(?c,?ce) ∧ bsg:successor(?r,?c) ∧ bsg:length(?r,?rl) ∧ math:eq(?ce,?rl) → bsg:lastSuccessor(?r,?c)

Table 8. Excerpt of format-specific rules for a limited PNG subset. Due to length considerations, the excerpt is limited to a set of rules capable of describing a PNG image to the level of chunk structures.

which contains information on image width and height.

5.1.3. Example deduction steps

For a given initial fact

```
bsg:source('root','oi2n0g16.png')    (13)
```

the deduction process tries to apply all rules to deduce new facts. In the first step, only the rules F1 and M1 are applicable, which yield the following new facts:

```
bsg:semantics('root','png:root')∧
bsg:start('root',0)∧
bsg:length('root',1432)              (14)
```

Again, the deduction process tries to apply all rules, this time on an increased set of facts. In step 2, the rules F2 and M4 yield the following:

```
bsg:type('root','bsg:structure')∧
bsg:firstSuccessor('root','_sc1')∧
bsg:semantics('_sc1','png:signature')∧
bsg:end('root',1432)                 (15)
```

The process of deduction is repeated until either no new facts can be deduced, or a computable predicate refutes a fact in a conclusion. The resulting facts from the reached fixed point describe a BSG instance for the PNG image oi2n0g16.png, which is part of the corpus and has a coverage of 1.0.

5.1.4. Result

After building a fitting set of rules with coverage of 1.0 for our corpus, we tested the set on all remaining PNG images from the PNG Test Suite. We obtained a coverage of 1.0 for 64 images, with the remaining 89 valid images having an average coverage of 0.79. Three corrupt images belonging to the test suite were excluded from the evaluation, as the fitting set of rules did not contain verifying rules for PNG-specific properties.

For a fitting set of rules over the entire PNG Test Suite, additional rules need to be included for palette handling (PLTE and sPLT chunks), transparency (tRNS chunk), background colour (bKGD chunk), textual data (tEXt and zTXt chunks) and other aspects. To estimate the effect of adding further rules, we added two preliminary rules for handling PLTE chunks and re-evaluated our rules on the corpus. We obtained a coverage of 1.0 for 78 images, with the remaining 75 valid images having an average coverage of 0.91.

During evaluation, the deduction process computed a fixed point and halted on all instances. Since errors may be present in a set of rules preventing a fixed point to be reached, a primitive approach on handling the Halting Problem is to place a limit on the iteration steps and abort the deduction beyond that limit. We discovered that the typical number of iterative steps required for our set of rules to reach a fixed point on valid PNG images ranges from 72 up to 170 steps. In case of the image file oi9n2c16.png, the number of iterative steps required was 3000+, as compressed image data is fragmented into bitstream segments with a length of 8 bit, each encapsuled into a separate IDAT chunk. This can be considered an extreme example, but demonstrates what is still considered legal in terms of the original specification. Since data format instances of other data formats such as Apple QuickTime movies have a more complex structure which requires an even higher number of iterations, the use of a semi-naive evaluation method for the deduction process as known from Datalog [37] is absolutely essential.

5.2. Documentation of an exploit

In this section, we give a practical example for documenting an exploit using Bitstream Segment Graphs. We have chosen the vulnerability CVE-2007-2365 [39] which utilizes a crafted PNG image to run malicious code. We use a specific version of this exploit that targets products from Adobe (Photoshop CS2, Photoshop CS3, Photoshop Elements 5) and Corel (Paint Shop Pro 11.20), but focus on the exploit section targetted for Adobe Photoshop CS2. The exploit itself was generated from an exploit generator which is available in C source code [40]. The exploit generator allows to select from two payloads, one starting the Windows Calculator, and another one binding a shell to port 4444, where we chose the former variant for annotation. For the documentation process, we consulted the PNG W3C specification [41] in addition to the crafted PNG image and the C source code of the exploit generator.

We use Apeiron introduced in Section 4.1.6 for creating a BSG instance as annotation on the crafted PNG image file. Splitting the file into its chunks according to the file specification, we arrive at the partial BSG instance shown in Figure 8. It consists of a PNG image signature and several *chunks*, which consist of a 32 bit length descriptor, a 32 bit type indicator, data of a length as indicated by the length descriptor, and a 32 bit CRC on the type and data field, if its length is non-zero.

At the current stage, the exploit contains three recognizable chunks, namely the "IHDR" chunk for describing basic information about the image, the "tIME" chunk for describing the last modification date, and the "pHYs" chunk describing the physical dimensions of pixels. The first two chunks are seemingly valid, describing an image of 509 pixel $\times$ 438 pixel with 256 indexed colors from a palette with standard values for the PNG compression, filtering and interlacing methods, which was last modified on 2007-04-15, 16:16:21 o'clock. The third "pHYs" chunk violates the PNG specification, as its length descriptor is expected to be 9 bytes, yet the ac-

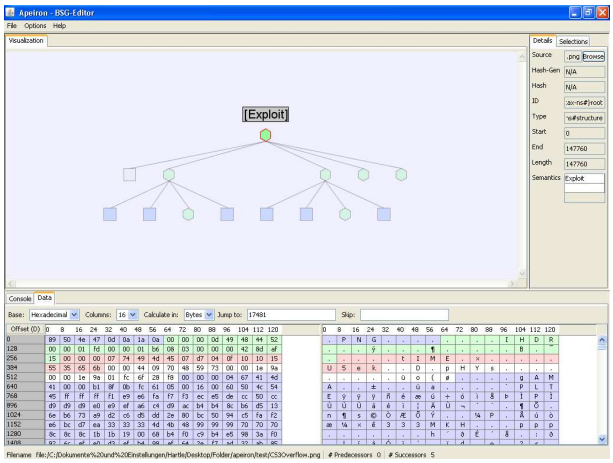

Fig. 8. Apeiron showing the crafted PNG image exploit structure with a signature, a "IHDR" chunk, a "tIME" chunk, an invalid "pYHs" chunk and junk data as its successors.

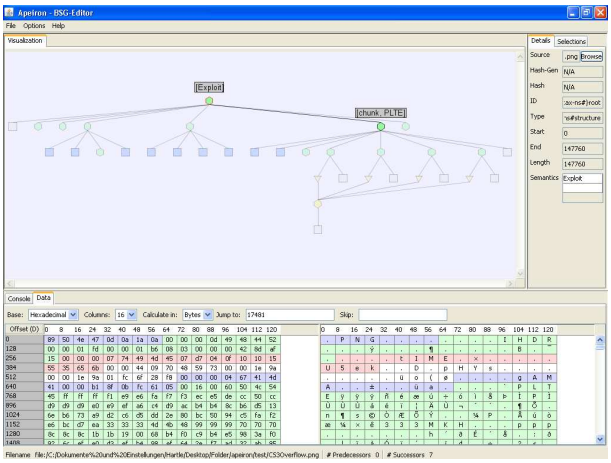

Fig. 9. Apeiron showing the crafted PNG image exploit structure as in Figure 8, adding a "gAMA" chunk and an invalid "PLTE" chunk to its successors when adjusting the "pYHs" length descriptor interpretation.

tual value is 0x4409 bytes. Taking this value for granted, the next location for a chunk contains plain invalid values, with a length descriptor value which is far beyond the actual file length (0xb67d641e), but which does not explain the actual execution of the Photoshop CS2 shellcode.

By either masking the length descriptor to the least significant byte or by assuming its specified length, we observe that the "pHYs" chunk is followed by a valid "gAMA" chunk, describing the gamma value of the purported image, a "PLTE" chunk, which is intended to describe the colors of the indexed palette for the image, shown in Figure 9. As each color is described from a red, green and blue color component each requiring 1 byte, an image with 256 colors should have a "PLTE" chunk no larger than 768 bytes, yet the actual value is 0x160060, which is well beyond the remaining file.

Interestingly, the exploit generator fragments its shellcode one byte at a time every three bytes in the "PLTE" chunk, corresponding to the red color component of the (invalid) indexed colors 1496+, which requires active reassembly through the targetted Photoshop CS2. In Figure 10, the annotation is shown for the first four bytes of the shellcode. We can therefore assume that the exploit at hand effectively overflows the buffer allocated for the red color component of indexed colors.

Another interesting aspect is that if we were to repeat the masking of the length descriptor to the least significant byte for the "PLTE" chunk as well, then the "PLTE" chunk is followed by an "IDAT" chunk, which normally would contain the filtered and compressed image data. We therefore assume that the "IDAT" chunk is actually carrying another exploit for a target application that performs the described masking of the chunk length descriptor for the invalid "pHYs" and "PLTE" chunks.

This example demonstrates the use of the Bitstream Segment Graph approach, as it allows a security expert to systemantically analyse the composition of data and document it appropriately.

6. APPLICATIONS IN IT SECURITY

6.1. Flaw Detection during Design

Just recently, the SANS and MITRE institutes performed a study in which "experts from more than 30 US and international cyber security organizations jointly released the consensus list of the 25 most dangerous programming errors" [42]. According to the consulted experts, "Improper Input Validation" was considered the most harmful.

Even though the topic is well known since the end of the 1980s [43], buffer overruns are still among the most harmful vulnerabilities. While many countermeasures have been researched and implemented, such as Java's ArrayOutOfBoundsException [44] or static and dynamic C code analysis [45], the data format itself has almost never been considered as a source of problems. Buffer overruns are usually considered to be programming mistakes, but they often result from the inherent complexity of data formats to be implemented.

At this point, BSGs can step in. For one, the systematic design process with tool-aided visualizations helps to make the structure and complexity of the data format apprehensive more easily. As well, it reduces the possibility of contradictory informations, which is often an issue in textual specifications. A self-consistent format specification eliminates one source of improper validation causing security flaws.

Second, it is hard to determine if an implementation conforms to a specification, but automatically generated parsers

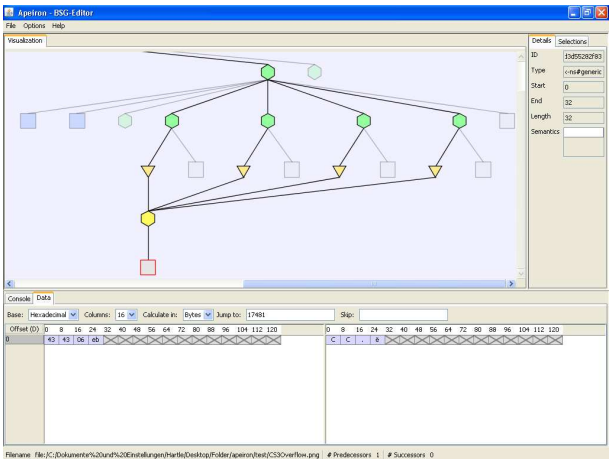

Fig. 10. Apeiron showing the start of fragmented exploit shellcode stored in the red color component of (invalid) indexed colors 1496+.

may often be infeasible or an implementation may already exist. This is where BSGs can be of help to identify extraordinary dangerous parts within a specification. To do so, we analyze a format specification regarding their level of redundancies. Redundant informations within format specifications may naturally lead to security vulnerabilities in a later implementation. Whenever there is e.g. several ways to conclude the length of a datum, a programmer might use one way to reserve memory and the second for the copy length, and a buffer overrun flaw is born. Too often, the programmer is not even aware of this change in informational context. For the programmer, the fast processing of data and an elegant code has the highest priority. By the formal description of parsing rules for BSGs with a underlying grammar, a specification designer can see and reason about the level of danger, i.e. the number of vulnerability-prone redundancies better. Hence, the specification can point out the most dangerous flaws to programmers from the start.

Third, data formats tend to be re-used in a way, other than their original objective. BSGs provide the means to analyze the behavior of nested data-formats and reason about the new situation. When a streaming-format is written to disk, for example, it might provide it's payload-size in the beginning, and the programmer would reserve memory accordingly. But due to the fact, that the filesystem provides the file's length as well, the payload can be read to EndOfFile, which may overrun reserved memory. The original format might not even have had a security relevant redundancy, but due to the nesting into a file, additional information has become available and a new vulnerability is introduced. The framework around BSGs provide a basis on which this kind of possible flaws can be analyzed in advance, but also for existing implementations.

6.2. Data Formats in Formal Security Validation

In the area of formal security validation, researchers as well as practitioners see themselves confronted with very complex computational and communication systems. Reasoning about security properties of a formal model for a full system can usually be considered infeasible. Therefore, the system model use in formal validation usually is based on an abstraction of the concrete system. Nevertheless, the completeness and soundness of the formal validation process demands a completely formally proven validation cycle requiring formalizations of these abstractions. However, most approaches for security verification neglect these abstractions and leave them out completely from their formal methodology. It is being assumed that the security expert, in order to be one, does not fail on assumptions done during the abstraction. A few however make these abstractions explicit. For example, [46], [47] and [48] present approaches on security preserving abstractions of system behavior based on alphabetic language homomorphisms. Still, the application of these formal concepts to the formalization of the abstraction of data formats is not straightforward. Ongoing research is trying to incorporate BSGs with the other to formalize the complete cycle from real to abstract systems and back in order to get evidence on the satisfaction of properties in the real system.

7. SUMMARY

In this article, we have presented the Bitstream Segment Graph and BSG Reasoning approaches for describing both data format instances and data formats. In contrast to related work, our approach is capable of handling arbitrary data format instances, providing the descriptive capabilities for structures, primitives, transcodes and fragments as well as functional dependencies. Moreover, we have shown significant limits from computational theory in our analysis, and chosen a feasible approach for modelling arbitrary data formats in general. We have presented examples of our approaches in use, describing a subset of the PNG data format as well as documenting a malicious crafted PNG image file targetting a vulnerability in Adobe Photoshop CS2. Last but not least, we have shown directions towards further applications and ongoing research regarding data format description in the area of formal security validation.

8. ACKNOWLEDGEMENTS

The authors would like to acknowledge the contribution of several students to our research and thank Arsene Botchak, Benno Kröger, Friedrich-Daniel Möller and Slaven Travar in strict alphabetic order for their master and diploma theses.

9. REFERENCES

[1] Michael Hartle, Daniel Schumann, Arsene Botchak, Erik Tews, and Max Mühlhäuser, "Describing Data Format Exploits using Bitstream Segment Graphs," in *Proceedings of The Third International Multi-Conference on Computing in the Global Information Technology (ICCGI)*, Athens, Greece, March 2008, IARIA, pp. 119–124, IEEE Press, New York, NY.

[2] *Reference Model for an Open Archival Information System (OAIS)*, vol. Blue Book, Consultative Committee for Space Data Systems, January 2002.

[3] Stephen L. Abrams and David Seaman, "Towards a Global Digital Format Registry," in *World Library and Information Congress: 69th IFLA GeneralConference and Council*, Berlin, August 2003.

[4] Adrian Brown, "File Format Registries and the PRONOM Service," in *ERPANET Seminar*, Vienna, 2003.

[5] John Marc Ockerbloom, *Mediating Among Diverse Data Formats*, Ph.D. thesis, Carnegie Mellon Computer Science, 1998.

[6] Michael Hartle, Friedrich-Daniel Möller, Slaven Travar, Benno Kröger, and Max Mühlhäuser, "Using Bitstream Segment Graphs for Complete Data Format Instance Description," in *Proceedings of The Third International Conference on Software and Data Technologies (ICSOFT)*, José Cordeiro, Boris Shishkov, Alphes Kumar Ranchordas, and Markus Helfert, Eds., Porto, Portugal, August 2008, Institute for Systems and Technologies of Information, Control and Communication, pp. 198–205.

[7] Michael Hartle, Arsene Botchak, Daniel Schumann, and Max Mühlhäuser, "A Logic-based Approach to the Formal Description of Data Formats," in *Proceedings of The Fifth International Conference on Preservation of Digital Objects (iPRES)*, London, United Kingdom, September 2008, The British Library, pp. 292–299.

[8] Wesley De Neve, Davy Van Deursen, Davy De Schrijver, Sam Lerouge, Koen De Wolf, and Rik Van de Walle, "BFlavor: A harmonized approach to media resource adaptation inspired by MPEG-21 BSDL and XFlavor," *EURASIP Signal Processing: Image Communication*, vol. 21, no. 10, pp. 862 –889, 11 2006.

[9] Davy De Schrijver, Wesley De Neve, Koen De Wolf, Robbie De Sutter, and Rik Van de Walle, "An optimized MPEG-21 BSDL framework for the adaptation of scalable bitstreams," *Journal of Visual Communication and Image Representation*, vol. 18, no. 3, pp. 217–239, 2007.

[10] Alexandros Eleftheriadis, "Flavor: a language for media representation," in *MULTIMEDIA '97: Proceedings of the fifth ACM international conference on Multimedia*, New York, NY, USA, 1997, pp. 1–9, ACM Press.

[11] Alexandros Eleftheriadis, "A Syntactic Description Language for MPEG-4," Contribution ISO/IEC JTC1/SC29/WG11 MPEG95/M0546, November 1995.

[12] O. Avaro, P. A. Chou, Alexandros Eleftheriadis, C. Herpel, C. Reader, and J. Signes, "The MPEG-4 Systems and Description Languages: A Way Ahead in Audio Visual Information Representation," *SP:IC*, vol. 9, no. 4, pp. 385–431, May 1997.

[13] Alexandros Eleftheriadis and Danny Hong, "Flavor: a formal language for audio-visual object representation," in *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, New York, NY, USA, 2004, pp. 816–819, ACM Press.

[14] Alexandros Eleftheriadis and Danny Hong, "XFlavor: Bridging bits and objects in media representation," in *Proceedings of the IEEE International Conference on Multimedia and Expo, 2002*, 2002, vol. 1, pp. 773–776.

[15] Alexandros Eleftheriadis, "The Benefits of Using MSDL-S for Syntax Description," Contribution ISO/IEC JTC1/SC29/WG11 MPEG96/M1555, November 1996.

[16] Anthony Vetro, Christan Timmerer, and Sylvain Devillers, *The MPEG-21 Book*, chapter Digital Item Adaptation - Tools for Universal Multimedia Access, pp. 243–281, John Wiley and Sons Ltd, 2006.

[17] M. Eisler, "RFC 4506: XDR: External Data Representation Standard," http://tools.ietf.org/html/rfc4506, January 2006.

[18] R. Srinivasan, "RFC 1831: RPC: Remote Procedure Call Protocol Specification Version 2," http://tools.ietf.org/html/rfc1831, August 1995.

[19] B. Callaghan, B. Pawlowski, and P. Staubach, "RFC 1813: NFS Version 3 Protocol Specification," http://tools.ietf.org/html/rfc1813, June 1995.

[20] ITU-T, "Recommendation X.680 (12/97) — Abstract Syntax Notation One (ASN.1): Specification of Basic Notation," ITU-T, Geneva, December 1997.

[21] ITU-T, "Recommendation X.691 (07/02) — ASN.1 Encoding Rules: Specification of Packed Encoding Rules (PER)," ITU-T, Geneva, July 2002.

[22] ITU-T, "Recommendation X.692 (03/02) — ASN.1 Encoding Rules: Specification of Encoding Control Notation (ECN)," ITU-T, Geneva, March 2002.

[23] ITU-T, "Recommendation X.509 (03/00) — The Directory: Public-key and attribute certificate frameworks," ITU-T, Geneva, March 2000.

[24] ITU-T, "Recommendation H.323 (06/06) — Packed-based multimedia communications systems," ITU-T, Geneva, June 2006.

[25] John Larmouth, *ASN.1 Complete*, Morgan Kaufmann, 1999.

[26] Michel Mouly, *CSN.1 Specification Version 2*, Cell & Sys, January 2002.

[27] ETSI, "Digital cellular telecommunications system (Phase 2+); Mobile radio interface signalling layer 3; General aspects (GSM 04.07 version 7.3.0 Release 1998)," December 1999.

[28] Protomatics, *The Transfer Syntax Notation One Specification*, Protomatics, Inc., 2006.

[29] James D. Myers and Alan Chappell, "Binary Format Description (BFD) Language," 2003.

[30] James D. Myers, Alan Chappell, Matthew Elder, Al Geist, and Jens Schwidder, "Re-integrating the research record," *Computing in Science and Engg.*, vol. 5, no. 3, pp. 44–50, 2003.

[31] Michael Beckerle and Alan Powell, "Data Format Description Language (DFDL) v1.0 Core Specification, Working Draft 032," http://forge.gridforum.org/sf/go/doc15262?nav=1, June 2008.

[32] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

[33] Michael Sipser, *Introduction to the Theory of Computation*, PWS Publishing, 1997.

[34] Charles H. Bennett, "Logical reversibility of computation," *IBM Journal of Research and Development*, vol. 17, no. 2, pp. 525–532, 1973.

[35] Tim Berners-Lee, "Notation 3," March 2006.

[36] Jeffrey Heer, Stuart K. Card, and James A. Landay, "Prefuse: A Toolkit for Interactive Information Visualization," in *Proceedings of ACM Human Factors in Computing Systems*, 2005, pp. 421–430.

[37] Jeffrey D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume II*, Computer Science Press, 1989.

[38] Willem van Schaik, "PngSuite - the official set of PNG test images," December 1998, http://www.schaik.com/pngsuite/pngsuite.html, last accessed 2008-01-02.

[39] MITRE, "CVE-2007-2365," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-2365, 2007.

[40] Marsu, "Photoshop CS2/CS3, Paint Shop Pro 11.20 .PNG File Buffer Overflow," http://milw0rm.com/exploits/3812, 2007.

[41] "Portable Network Graphics (PNG) Specification (Second Edition): Information technology – Computer graphics and image processing – Portable Network Graphics (PNG): Functional specification. ISO/IEC 15948:2003 (E)," November 2003.

[42] Bob Martin, "CWE/SANS TOP 25 Most Dangerous Programming Errors," http://www.sans.org/top25errors/, January 2009.

[43] Mark W. Eichin and Jon Rochlis, "With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988," in *Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy*, Oakland, Ohio, 1989, pp. 326–343, IEEE Computer Society.

[44] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The Java Language Specification (3rd Edition)*, Addison Wesley, 2005.

[45] David Wagner, Jeffrey S Foster, A. Eric Brewer, and Alexander Aiken, "A First Step towards Automated Detection of Buffer Overrun Vulnerabilities," in *Network and Distributed System Security Symposium*, San Diego, California, USA, February 2000, pp. 3–17.

[46] Sigrid Gürgens, Peter Ochsenschläger, and Carsten Rudolph, "Abstractions Preserving Parameter Confidentiality," in *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS) 2005*, Milan, Italy, September 2005, vol. 3679 of *Lecture Notes in Computer Science*, pp. 418–437, Springer Verlag.

[47] Peter Ochsenschläger, Jürgen Repp, and Roland Rieke, "Abstraction and composition: a verification method for co-operating systems," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 12, pp. 447–459, 2000.

[48] Andreas Fuchs, Sigrid Gürgens, and Carsten Rudolph, "On the Security Validation of Integrated Security Solutions," in *Proceedings of IFIP Sec 2009*, 2009.