Static Analysis for Checking Data Format Compatibility of Programs

Pranavadatta Devaki[1] and Aditya Kanade[2]

1 **IBM Research India**
2 **Indian Institute of Science**

Abstract

Large software systems are developed by composing multiple programs. If the programs manipulate and exchange complex data, such as network packets or files, it is essential to establish that they follow compatible data formats. Most of the complexity of data formats is associated with the headers. In this paper, we address compatibility of programs operating over headers of network packets, files, images, etc. As format specifications are rarely available, we infer the format associated with headers by a program as a set of guarded layouts. In terms of these formats, we define and check compatibility of (a) producer-consumer programs and (b) different versions of producer (or consumer) programs. A compatible producer-consumer pair is free of type mismatches and logical incompatibilities such as the consumer rejecting valid outputs generated by the producer. A backward compatible producer (resp. consumer) is guaranteed to be compatible with consumers (resp. producers) that were compatible with its older version. With our prototype tool, we identified 5 known bugs and 1 potential bug in (a) sender-receiver modules of Linux network drivers of 3 vendors and (b) different versions of a TIFF image library.

1998 ACM Subject Classification D.2.4 Software/Program Verification; D.2.12 Interoperability

Keywords and phrases Data format compatibility, producer-consumer/version-related programs

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Large software systems are developed by composing multiple programs. For the composed system to work as desired, it is essential that the component programs are compatible with each other. Compatibility being a fundamental requirement in system integration, various techniques for analyzing compatibility have been proposed. Finite state machines have been used for checking compatibility of programs with respect to temporal aspects of method calls or action sequences [14], message sequences [17, 33], and typed accesses to sequential data [15]. When a component program evolves, the two versions can be compared to extend compatibility guarantees of the older version to the newer version. Incompatibilities in component upgrades have been identified by checking implication between logical summaries of observed behaviors of the two versions [26]. Automata learning and model checking have been combined for determining substitutability of programs [32, 11].

In many software systems, the component programs manipulate and exchange complex data like network packets or files. Checking compatibility of such programs involves comparing the data formats followed by them. A data format describes types and constraints on values of the data elements. To be *type-compatible*, the programs must use compatible types while accessing the same data elements. Type incompatibility between programs is observed frequently in practice. As an example, consider the bug report [1] which describes a type incompatibility between Atheros sender-receiver pair. For MAC frames that carry payload,

Conference title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 1–12

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Figure 1 Mis-interpretation of a MAC ACK frame by an Atheros receiver: (a) Layout of the incoming ACK frame and (b) The type-incompatible layout computed by the receiver

the data following the control data is 4-byte aligned. ACK frames do not carry payload and their control data is followed immediately by a CRC checksum. Figure 1(a) shows an ACK frame received by the receiver. The receiver incorrectly computes the starting index of CRC for the ACK frame as 12 by computing the next 4-byte aligned index after 10 (end-index of the control data). As shown in Figure 1(b), it reads bytes $12 - 15$ as an integer containing the CRC. This access overlaps with the last two bytes of the integer stored at offsets $10 - 13$ and offsets $14 - 15$ of the remaining data of the incoming frame.

Another reason for incompatibilities is the complexity of constraints over data values. The formats of network packets, files, and images support a number of variants. For example, an image can be an RGB or a grayscale image. A MAC frame can be a data, control, or management frame. To distinguish between the variants, the format prescribes that the *producer* program should constrain specific elements of the data to pre-determined values. For example, a MAC frame is marked as a control frame by setting bits $2 - 3$ of the frame control field to binary 10. A *consumer* program should check the same elements for equivalent constraints to identify the variants. Thus, in addition to types, the compatibility between a producer-consumer pair also depends on whether *the consumer accepts all variants or only some variants* of the data generated by the producer. The latter case can give rise to compatibility errors. As described in the bug report [2], an Intel receiver treats frames of length < 30 as undersized (corrupt) and drops them. However, the minimum frame length depends on the variant the frame belongs to. The check against the hard-coded value of 30 results in frames belonging to a variant being dropped. As a result, repeated attempts at communication with a sender, that produces them, end in timeout of an `ssh` session.

The constraints required to identify variants of data are typically encoded in a pre-defined region of data, called the meta-data or *header*, *e.g.*, packet and file headers. We call the rest of the data as the contents. A header determines interpretation of the contents. The mismatch in processing of headers is therefore a key reason for data format incompatibilities (*e.g.*, see bug reports [1, 2, 3, 4, 5]). As the bug reports indicate, if incompatibilities are not detected during system integration, they can cause a variety of runtime errors like incorrect output [4], rejection of input [5], data corruption [1, 3], and non-termination [2]. In this paper, we analyze compatibility of programs operating over headers of network packets, files, images, etc. The headers are usually of fixed-length whereas the contents can be unbounded. We consider headers of fixed-length. Variable-length headers are also common but their lengths commonly vary over a finite set and are thus a simple extension of fixed-length headers.

As the data format specifications are not available to us, we design a static analysis of sequential C programs to infer them. The approach presented in this paper applies to any *fixed-length data* (including but not limited to headers) that programs may exchange, *e.g.*, serialized data structures. We shall simply refer to these as (fixed-length) data and their formats as data formats. A *data format* is formalized as a finite set of pairs of the form (g, ℓ), called *guarded layouts*, where g is a symbolic constraint over data elements, called the *guard*, and the *layout map* ℓ maps types to sets of offsets into the fixed-length data. The guard represents a set of data values and the layout map gives their type. We formalize compatibility relations in terms of guarded layouts for (a) producer-consumer programs

(b) two versions of producer (or consumer) programs. A compatible producer-consumer pair is free of type incompatibilities and mismatch in processing of variants of data. The version-related programs are executed individually without any direct data exchange between them. Nevertheless, they are expected to work with the same programs. A backward compatible producer (resp. consumer) is guaranteed to be compatible with consumers (resp. producers) that were compatible with its older version.

We are not aware of any approach for compatibility checking that models both types and constraints in data format specifications. The approaches [17, 33, 15] model types of data elements but *not* constraints over them. As noted earlier, constraints are essential for accurate description of formats and to find subtle compatibility bugs. The data description languages PacketTypes [27], DataScript [8], and PADS [18] do permit modeling of both types and constraints. LearnPADS++ [35] even generates data descriptions automatically from the example data. However, their goal is to generate parsers, type checkers, etc. for data formats. In contrast, our goal is to infer the formats from C programs and use them to check program compatibility. Further, the above compatibility checking approaches consider programs which access data *sequentially* from FIFO channels [17, 33] or as data streams [15]. In our case, a program accesses the data as an in-memory data structure. These accesses are *not* necessarily sequential. Unlike the other approaches, we formalize compatibility of *version-related* producer/consumer programs as well.

We have applied our prototype tool to check data format compatibility of several programs. The producer-consumer case studies include sender-receiver modules for IEEE 802.11 MAC frame headers of Linux network drivers of 3 vendors, namely, Atheros, Intel, and Intersil obtained from the Linux distributions. The version-related case study involves 48 functions operating over meta-data of TIFF files from 2 pairs of different versions of libtiff [6], a popular image processing library. We identify 5 known bug and 1 potential bug in these programs. In two cases, bug fixes are available for known bugs. Our static analysis confirms that the fixes resolve the compatibility issues. In summary,

- The paper defines data formats as guarded layouts for fixed-length data like packet and file headers. The formats are inferred by static analysis of C programs (Section 3).
- Compatibility relations over data formats and runtime guarantees provided by them are formalized for producer-consumer and version-related programs (Section 4).
- The approach is implemented and used for finding bugs and proving compatibility of real-world programs (Section 5). Finally, the work is compared with related approaches (Section 6) and future work is discussed (Section 7).

2 Overview

In this section, we present an overview of our approach with examples. Consider the programs producer and consumer given in Figure 2. These are inspired by sender and receiver modules that prepare and interpret network packet headers. We shall consider a modified version of producer to illustrate compatibility of version-related programs.

The program producer prepends a header to the frame received through buf. The user designates buf as its output variable. It can encode three variants of headers declared as struct hdr, qhdr, and lhdr. It accesses buf by overlaying buf with all of them. The program consumer reads the header from the (user-designated) input variable buf. For each of the designated variables, the user provides the starting offset and length of the sequential data that can be accessed through it. In the case of pointer variables, the accesses are by dereferencing the pointers. In this example, buf points to offset 0 in each program. The

```
1 int devmode, qosmode; // global variables configured by other components
2 struct hdr { short tods,fromds,type; char restofhdr[14]; int len; };
3 struct qhdr { short tods,fromds,type,qos; char restofhdr[14]; int len; };
4 struct lhdr { short tods,fromds,type; char restofhdr[20]; int len; };
```

```
5 int producer(char* buf, short qos, int blen) {
6   struct hdr* hd = (struct hdr*) buf;
7   struct qhdr* qhd = (struct qhdr*) buf;
8   struct lhdr* lhd = (struct lhdr*) buf;
9
10  if(devmode == 0) {
11    hd->tods = 0; hd->fromds = 1;
12    if(qosmode == 1) {
13      qhd->type = 1;
14      qhd->qos = qos;
15      qhd->len = blen + 26;
16    } else {
17      hd->type = 0;
18      hd->len = blen + 24;
19    }
20  } else if(devmode == 1) {
21    lhd->tods = lhd->fromds = 1;
22    lhd->type = 0;
23    lhd->len = blen + 30;
24  } else return ERR; // error return
25
26  return 0; // normal return
27 }
```

```
28 int consumer(char* buf) {
29   int len = 0;
30   struct hdr* hd = (struct hdr*) buf;
31
32   if(hd->len < sizeof(struct ldr))
33     return ERR; // error return
34
35   if(hd->tods == 1 && hd->fromds == 1)
36     return ERR; // error return
37
38   if(hd->tods == 0 && hd->fromds == 1)
39     len = hd->len;
40
41   return len; // normal return
42 }
```

Figure 2 Example of a producer-consumer pair of programs

other interface variables and global variables of producer are initialized by its environment and are not part of the data being exchanged. The length of the data is given to be 30. It can accommodate any of the three variants of the header. The user classifies the return statements into error and normal return as annotated in Figure 2.

The two programs do not run on the same machine. The actual transportation of data between them is handled by I/O routines which are not part of the analysis. Since the data exchange may happen across platforms, compatibility issues like differences in endian-ness may arise at lower-levels of abstraction. These issues are outside the scope of this paper.

Static inference of data formats. The two programs being checked for compatibility can use different type declarations and variable names, *e.g.*, programs developed by different vendors. Using structure declarations as data formats to check compatibility is not sufficient as they do not capture the constraints over data elements. Moreover, since data is often manipulated using pointers and type casts, structure declarations do not always indicate the layout map used to access the data. Hence, we infer data formats using static analysis. We represent data formats in a program-independent manner so that they can be compared. We consider the *primitive types* like characters and integers instead of user-defined types like hdr and qhdr. Thus, our layout maps are at a lower-level of abstraction than the types appearing in the programs. Further, we follow a *uniform naming convention of data elements* based on their offsets into the data and sizes. The guards are defined over these uniformly-named data elements. Since we want guarded layouts of output of a producer, we perform a forward analysis. Analogously, we perform a backward analysis of consumers.

In Figure 2, lines 6-8 type-cast and create aliases to buf. There are 3 paths in producer that reach the normal return at line 26. The return at line 24 is an error return and the output produced when this return is reached is not considered to be a valid output. Consider the path P through the first 2 if-branches (lines 10 and 12) and ending at the normal return.

- The data format (set of guarded layouts) of producer is $\{(g_1, \ell_1), (g_2, \ell_2), (g_3, \ell_3)\}$ where
  $g_1 : d[0:1] = 0 \wedge d[2:3] = 1 \wedge d[4:5] = 1$
  $\ell_1 : [short \mapsto \{0,2,4,6\}, int \mapsto \{22\}, x \mapsto \{8,\ldots,21,26,\ldots,29\}]$
  $g_2 : d[0:1] = 0 \wedge d[2:3] = 1 \wedge d[4:5] = 0$
  $\ell_2 : [short \mapsto \{0,2,4\}, int \mapsto \{20\}, x \mapsto \{6,\ldots,19,24,\ldots,29\}]$
  $g_3 : d[0:1] = 1 \wedge d[2:3] = 1 \wedge d[4:5] = 0$
  $\ell_3 : [short \mapsto \{0,2,4\}, int \mapsto \{26\}, x \mapsto \{6,\ldots,25\}]$
- The data format of consumer is $\{(g_4, \ell_4)\}$ where
  $g_4 : d[0:1] = 0 \wedge d[2:3] = 1 \wedge d[20:23] \geq 30$
  $\ell_4 : [short \mapsto \{0,2\}, int \mapsto \{20\}, x \mapsto \{4,\ldots,19,24,\ldots,29\}]$

Figure 4 Data formats inferred from producer and consumer programs of Figure 2

The set of typed-accesses to the data through the pointers along this path (see Figure 3) determine the layout map. The layout map according to these accesses is given as ℓ_1 in Figure 4. The type x is a special type, called *don't care*, which indicates that there was no access to the byte. It allows us to detect if two programs make non-identical accesses.

Now, consider the symbolic path condition and program state at the end of the path. As the variables devmode, qosmode, qos, and blen are not among the designated variables, we forget constraints over them. This is because these variables are not visible to the consumer and its processing of the data cannot depend on them. In order to compare guards across programs, we uniformly name the fields accessed through hd, qhd, and lhd as $d[i:j]$ where i is the starting offset of the field and $j = i+k-1$ with k equal to size of the field. The guard g_1 corresponding to the layout map ℓ_1 is shown in Figure 4. As the values of arguments qos and blen are unconstrained, the values of data elements $d[6:7]$ and $d[22:25]$ assigned from them also remain unconstrained and are not shown in g_1. For consumer, there is only one path that leads to a normal return (line 41). The data formats of the programs are given in Figure 4.

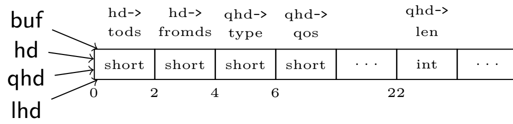


Figure 3 Pictorial view of the typed-accesses along path P to the data prepared by producer

Producer-consumer compatibility. The compatibility of a pair of programs can be checked using the inferred data formats. One of the program acts as a producer and another as a consumer. The same program may act as a producer or a consumer depending on the context. Programs like network protocol implementations exchange data interactively and the complete analysis of compatibility between them additionally requires analysis of temporal ordering of data exchanges. Our computation of data formats yields the set of guarded layouts exchanged from all possible protocol-states (if any) of such a program. This suffices to identify *data format* incompatibilities.

As a compatibility question, we can ask (1) whether the consumer accesses data elements with types compatible with those of the producer? The answer for our example is no, as $g_1 \wedge g_4$ is satisfiable but $\ell_4(int) \not\subseteq \ell_1(int)$. As the conjunction of guards is satisfiable, there is some datum generated by producer that consumer accepts. However, consumer accesses offsets 20-23 of the datum as an *int* unlike producer. We can also ask (2) whether consumer accepts *every* output of producer (in a type-compatible manner)? The answer is no, as $g_2 \not\rightarrow g_4$, since $d[20:23]$ (denoting hd−>len) is unconstrained in g_2. We shall formalize the above questions

respectively as *weak* and *strong* compatibility requirements.

Compatibility of version-related programs. Let us now consider a newer version of producer by removing lines 20-23. Can we substitute the newer version in place of the older version? The answer in this case is affirmative. The data format of the newer version is $\{(g_1, \ell_1), (g_2, \ell_2)\}$. Thus, the newer version produces a subset of the outputs produced by the older version and associates the same layout maps to the common outputs. We call the newer version of the producer *backward* compatible in this case. If a consumer is type-safe (weakly compatible) with the older version of producer, then it will be type-safe with the newer version also. We shall similarly infer how strong compatibility is preserved by a backward compatible producer. Backward compatibility is also defined for consumers.

False positives and negatives. The guards inferred by a sound static analysis can be over-approximations of conditional accesses to the data by the programs. The over-approximations in points-to information can also cause imprecision. A compatibility result with imprecise formats can be a false positive or negative (discussed in Section 4). Similar limitations apply to related approaches [26, 15]. In the case of [26], the limitation stems from implication checks between dynamically inferred constraints which can be unsound (under-approximate) or imprecise (over-approximate). The automata computed in [15] are over-approximate. Therefore, the language containment between such automata may result in a false positive or negative. We control the false positives and negatives for our case studies by designing a sufficiently precise analysis with a suitable choice of abstract domain.

3 Static Inference of Data Formats

Since C programs routinely use pointers, we perform a points-to analysis to identify (abstract) memory locations that a pointer may point to. The precision of points-to information is important to us to avoid coarse over-approximations of the analysis results. We therefore perform a flow-sensitive may points-to analysis based on [16]. Points-to analysis and the memory model used are discussed further in the extended version [7]. We first formalize the notion of data formats for fixed-length data.

▶ **Definition 1.** A *data format* is a finite non-empty set of pairs of the form (g, ℓ), called *guarded layouts*, where g is a symbolic constraint over data elements, called the *guard*, and the *layout map* ℓ maps primitive types to subsets of the set OL of offset-based locations.

Intuitively, every guarded layout represents a variant supported by the data format and encoded in the program. For a layout map ℓ and a type t, if $k \in \ell(t)$ then it indicates that a *data element* of type t resides at offset k and it occupies offsets $\{k, \dots, k + sizeof(t) - 1\}$. Every offset-based location in OL must belong to the set of offsets occupied by exactly one data element. The type of an offset is inferred only if it is accessed in the program. Unaccessed bytes are assigned the *don't care* type x. As discussed in Section 2, if an element of type t resides at offset i then it is named as $d[i : j]$ where $j = i + sizeof(t) - 1$. The guards are symbolic constraints over these uniformly-named data elements.

In order to compute a sound over-approximation of the set of guarded layouts, we perform abstract interpretation [13]. Our analysis computes layout maps and the corresponding sets of reachable states at every control location in the program. The reachable state-sets are represented using an abstract domain. The abstract domain is defined over uniformly-named data elements as well as program variables which do not refer to the offset-based locations. This is essential to capture all possible dependences between program variables. We call these as *extended guards*. The data format (the set of guarded layouts) is obtained by restricting

the extended guards to the data elements, at appropriate control locations. For a producer, the format is obtained at program exits whereas for a consumer it is obtained at the program entry. Due to space constraints, we discuss analysis of producers only.

Abstract domains. Let $\mathcal{G} = (G, \sqsubseteq_G, \sqcup_G, \sqcap_G, \bot_G, \top_G)$ be an abstract lattice to represent extended guards, with $\sqsubseteq_G$ as the partial order, $\sqcup_G$ as join, $\sqcap_G$ as meet, and $\bot_G$, $\top_G$ resp. as bottom and top elements. Since the values of data elements can range over infinite domains, we require $\mathcal{G}$ to be equipped with a widening operator. Let $\mathcal{L} = (L, \sqsubseteq_L, \sqcup_L, \sqcap_L, \bot_L, \top_L)$ be the *finite* lattice of layout maps of a fixed length. We have $\ell \sqsubseteq_L \ell'$ if $\ell(x) \supseteq \ell'(x)$ and for each primitive type $t \neq x$, $\ell(t) \subseteq \ell'(t)$. Let $D = \mathcal{G} \times \mathcal{L}$ be the domain of layout maps with extended guards. We denote an extended guard with $\hat{g}$ and a guard restricted to only data elements by g.

Transfer functions. Let $(\hat{g}, \ell) \in D$ be the abstract element before the current control location. If the statement contains a pointer r that may point to multiple memory locations then, for each of the memory locations, we create a *separate* transformed guarded layout. Thus, $(\hat{g}, \ell)$ may map to a *set* of guarded layouts. This is to keep the guarded layouts precise individually at the cost of increase in the number of guarded layouts. The guarded layout $(\hat{g}, \ell)$ is updated by considering the *same memory location* simultaneously in transfer functions of *both* $\hat{g}$ and ℓ. Within a loop, we consider all memory locations simultaneously. The transfer functions for extended guards are defined by the chosen abstract interpretation over the domain $\mathcal{G}$. The choice of $\mathcal{G}$ is an implementation issue.

The transfer functions for layout maps are now presented. The types of offset-based locations are inferred based on accesses to them. For a producer, we infer the layout maps by considering both read and write accesses to offset-based locations. The offset-based locations can be accessed through any valid C expression involving a user-designated variable or pointer (and aliases to it). The space of syntactic expressions is large and we consider only some representative cases. Let us denote a pointer by r, a structure (or union) by s, a structure field by f, a type by t, and a scalar variable by v. Let us denote the points-to map at a control location by PM and the compiler assigned type of an expression e by $\mathsf{typeof}(e)$. Consider the commonly used expressions v, $*r$, $s.f$, and r->f. Let E be an expression occurring at the current control location and $\mathsf{subexp}(E)$ be the set of subexpressions of E which fall into the above expression classes. An incoming layout map ℓ is potentially updated by each subexpression $e \in \mathsf{subexp}(E)$. Operators in E do not play any role in updating of layout maps. In the case of v and $s.f$, an incoming layout map is updated only if v and s are user-designated variables. In the case of $*r$ and r->f, we check whether $PM(r) \cap OL$ is empty or not. If it is empty then the expression is ignored.

For a suitable expression e, the offset-based location k accessed by it is computed. For v, it is the user-designated offset. For $*r$, it is an offset-based location in $PM(r)$. For $s.f$ and r->f, let b be the user-given offset of (the first field of) s or an offset-based location in $PM(r)$. Suppose the type of s is `struct T` and of r is pointer to `struct T`. The offset-based location k corresponding to $s.f$ or r->f is obtained by incrementing b by the cumulative sum of sizes of fields preceding the field f in `struct T`. The type of k in the layout map ℓ is changed to $t = \mathsf{typeof}(e)$ as follows: $\ell' := \ell[t \mapsto \ell(t) \cup \{k\}, x \mapsto \ell(x) \setminus \{k, \ldots, k + sizeof(t) - 1\}]$. If an offset ends up getting assigned to multiple data elements in a layout map, then we have found a potential *type error* and the analysis stops.

For example, the incoming set of guarded layouts at Line 11 of Figure 2 consists of a single guarded layout $(\hat{g}, \ell)$ where $\hat{g}$ is $\mathsf{devmode} = 0$ (corresponding to the predicate of the conditional at Line 10) and ℓ is $\bot_L$. Note that $\hat{g}$ is an extended guard and hence is not

restricted to constraints over data elements only. ℓ is $\bot_L$ as no access to data is made yet in the program. At Line 11, ($\hat{g}$, ℓ) is transformed into ($\hat{g}'$, ℓ'), where $\hat{g}'$ is devmode $= 0 \wedge d[0:1] = 0 \wedge d[2:3] = 1$ and ℓ' is $[short \mapsto \{0, 2\}, int \mapsto \{\}, x \mapsto \{4, \ldots, 29\}]$, to capture the effect of the assignment statements and the accesses to data at offsets 0 and 2 (through hd−>tods and hd−>fromds).

The overall analysis. Based on our experience with the case studies, we realize that maintaining branch-correlations is useful. We therefore perform a forward path-sensitive analysis by taking union of sets of guarded layouts at join points (outside loops). For a loop, the incoming set of guarded layouts is treated as an ordered set. The termination of the analysis is ensured by avoiding (potentially unbounded) growth in the number of guarded layouts and using the widening operator of domain $\mathcal{G}$. For statements within a loop, the transfer functions map a guarded layout to a single transformed guarded layout and at join points, we take element-wise join of the ordered sets of guarded layouts.

We compute procedure summaries bottom-up and use them at procedure call sites. Let $(\hat{g}, \ell)$ be an extended guarded layout at a call site. The constraints over variables or memory locations in the calling context that are written into by the callee are forgotten from $\hat{g}$. If a formal parameter is not written into by the callee then its constraints in the procedure summary are transferred to the corresponding actual. The guard $\hat{g}$ is combined with every guard $\hat{g'}$ in the procedure summary as $\hat{g} \sqcap_G \hat{g'}$. The matching layout map is obtained by $\ell \sqcup_L \ell'$. Finally, the assignment of the return value to the LHS of the call (if any) is processed. The guard is then restricted to variables in the calling context.

4 Compatibility Relations

We assume that precise data format specifications, in the form of sets of guarded layouts, are available. We define the compatibility relations over them and discuss the effect of *imprecision* in the format specifications, obtained by static analysis, subsequently.

Consider two layout maps ℓ and ℓ' defined over the same set of offset-based locations. The layout map ℓ is *compatible* with ℓ', denoted by $\ell \preceq \ell'$, if $\ell(x) \supseteq \ell'(x)$ and for every $t \neq x$, $\ell(t) \subseteq \ell'(t)$. Equality of layout maps ($=$) is obtained by replacing $\subseteq$ and $\supseteq$ with $=$ in the above definition. If $\ell \preceq \ell'$, then every datum produced with layout map ℓ' can be consumed at runtime with layout map ℓ without type incompatibilities. Given a set G of guarded layouts of a program, the program is *locally compatible* if the guards in G with different layout maps are pairwise disjoint. If a datum can be produced with different layout maps, consumers cannot know which layout map to use for it. In the following discussion, we assume that the programs are locally compatible.

Producer-consumer programs. We define two compatibility relations between producer-consumer programs. Weak compatibility guarantees that if a consumer consumes a datum produced by a producer, it accesses it using a compatible layout map. Strong compatibility ensures, in addition, that every datum produced by a producer is consumed by the consumer. Strong compatibility implies weak compatibility.

▶ **Definition 2.** Let G and G' be sets of guarded layouts representing the data formats of a producer program P and a consumer program C respectively. P is *weakly compatible* with C, denoted by $WC_{\preceq}(P, C)$, if for every $(g, \ell) \in G$ and $(g', \ell') \in G'$, if $g \wedge g'$ is satisfiable, then $\ell' \preceq \ell$. P is *strongly compatible* with C, denoted by $SC_{\preceq}(P, C)$, if for every $(g, \ell) \in G$, there exists a $(g', \ell') \in G'$ such that $g \Rightarrow g'$ and $\ell' \preceq \ell$.

Stricter variants of the above definitions, $WC_{=}(P, C)$ and $SC_{=}(P, C)$, are obtained by replacing layout compatibility ($\preceq$) with layout equality ($=$) in them. These are useful to identify whether the two programs access the same set of offsets or not.

Version-related programs. Our definition of backward compatibility allows a newer version of a producer to initialize more offsets of the data, but restricts it to produce a subset of values taken by the data elements. If the producer initializes less offsets then they remain unconstrained and the set of data values produced is more. For a newer version of a consumer, consuming more data values is allowed, but reading from more offsets of the data is disallowed.

▶ **Definition 3.** Let G and G' be sets of guarded layouts representing the data formats of two versions P and P' of a producer or two versions C and C' of a consumer.

1. The producer P' is *backward compatible* with P, denoted by $BC_{\preceq}(P', P)$, if for every $(g', \ell') \in G'$, there exists $(g, \ell) \in G$ such that $g' \Rightarrow g$ and $\ell \preceq \ell'$.
2. The consumer C' is *backward compatible* with C, denoted by $BC_{\preceq}(C', C)$, if for every $(g, \ell) \in G$, there exists $(g', \ell') \in G'$ such that $g \Rightarrow g'$ and $\ell' \preceq \ell$.

Stricter versions of backward compatibility can be obtained by using layout equality ($=$) in place of layout compatibility ($\preceq$). A backward compatible producer can be substituted for its older version while preserving the compatibility relations of the older version. A backward compatible consumer preserves strong compatibility but does *not* necessarily preserve weak compatibility. The precise formulation of compatibility relations preserved by backward compatible programs and their proofs are discussed in the extended version [7].

Analysis of false positives and negatives. The guards obtained by static analysis can be over-approximations of the precise constraints associated with a layout map. Both local and weak compatibility definitions require checking satisfiability of $g \wedge g'$. Since the check is done using the over-approximated guards, the satisfiability can be spuriously true. Thus, compatibility of layout maps of disjoint guards may have to be checked and it can fail. Hence, local and weak compatibility checks are sound with respect to guards, but can result in false positives. The layout maps can also over-approximate the sets of typed-accesses. Thus, compatibility check over them may result in a false positive or negative, due to set inclusion over offset-sets used in layout map compatibility. The strong and backward compatibility require checking implication between guards. Because of over-approximation, this check can result in either a false positive or negative.

5 Experimental Evaluation

This section summarizes the experimental results. We have prototyped our approach in OCaml using CIL [29] and the octagon library [28]. The complexity of most of the operations over octagons is cubic in the size of the octagon.

Producer-consumer programs. We analyze sender-receiver modules of Linux (version 2.6.33.3) wireless LAN drivers of three vendors, namely, Atheros, Intel, and Intersil. They operate over IEEE 802.11 MAC frame headers. We bit-blast some fields of the headers. In addition, we consider a pair of programs from libtiff [6]: set/get routines that allow clients to get and set TIFF directory attributes. In all, 9 programs – 3 producers and 6 consumers – were analyzed. Of these, two consumers were with fixed versions of programs. These programs are a few hundreds of lines in size and the number of guarded layouts range from a few tens to over 100. All programs were analyzed in about 75 seconds. All variants of weak

No.	Programs	Bug Description	Failed Check
1.	Atheros-Atheros	Consumer reads offsets not written by the producer [1]	$WC_{\preceq}$
2.	Intersil-Intel	Valid packets are discarded by the consumer [2]	$SC_{\preceq}$
3.	TIFF set-get	An offset is written as a short and read as an int [3]	$WC_{\preceq}$
4.	tiffwrite (7 programs)	New version of the producers start producing images with field_planarconfig = 0 [4]	$BC_{\preceq}$
5.	tiffread (14 programs)	New version of the consumers access more offsets	$BC_{\preceq}$
6.	tiffcp	New version of the consumer accesses more offsets [5]	$BC_{\preceq}$

Figure 5 Description of data format compatibility bugs found by our tool

and strong compatibility were checked for 6 producer-consumer pairs. Figure 5 describes the bugs found by our tool. Both Atheros and Intersil-Intel bugs were discussed in the Section 1. The TIFF set-get routines fail the weak compatibility check as they access an offset with different types. All compatibility checks were evaluated in about 26 seconds.

Version-related programs. We analyze the TIFF directory manipulation routines of two versions, 3.7.4 and 3.9.4, of libtiff [6] and two versions, 3.5 and 3.6, of the TIFF copy tool tiffcp. In all, 42 libtiff routines (26 producers from tiffwrite module and 16 consumers from tiffread module) and 6 tiffcp routines (both as producers and consumers) were analyzed in about 7 minutes. These versions are several hundreds of lines of code in size and generate tens to over 150 guarded layouts. The newer versions of the producer routines of tiffwrite module produce images with field_planarconfig = 0, which the older version did not produce. Some of the tiffread routines of the newer version access more offsets of the data than the older version. This bug was not reported earlier. Similarly, one tiffcp consumer accesses more offsets than its older version. All compatibility checks were evaluated in about 21 seconds.

False positives and false negatives. The backward compatibility checks for tiffread and tiffcp fail due to two reasons: (1) For some guard g of the older version of a consumer, there is no guard g' in the newer version such that $g \implies g'$. (2) The newer consumer reads more offsets. The second reason is a true positive (bug). The first reason is a false positive for these examples. The newer consumers have case splits on a particular value. The associated layout maps are equal and if we take union of the guards then g implies the union. However, as union in octagonal domain is not precise, we do not perform it by default. There were no false negatives in the case studies.

6 Related Work

Compatibility checking. In Section 1, we have compared our work with related work on compatibility of producer-consumer programs [17, 33, 15]. Our intuition that a backward-compatible producer should produce less and a backward-compatible consumer should consume more is similar to behavioral subtyping [24] over class hierarchies. The refinement of interface automata [14] and implications between pre/post conditions in [26] model similar constraints for version-related programs. Substitutability analysis [32, 11] allows the newer version to exhibit more behaviors provided it preserves system-level safety properties.

Representation dependence testing [20] evaluates *forward compatibility* of consumers by simulating a large class of producers by sequentially composing a specification program and its algorithmically derived inverse. Compatibility tests have been generated from behavioral models of components which include sequences of actions and constraints over data being

exchanged [25]. However, the models are synthesized from observed behaviors of programs and do not identify type incompatibilities. Fingerprint generation [9] generates inputs that take two implementations of a protocol to different states. They symbolically execute binaries and use decision procedures to generate inputs.

Data format specification and inference. Data description languages [27, 8, 18] permit more expressive specifications than our guarded layouts, *e.g.*, they provide type constructors for unbounded data. However, their goal is to generate supporting tools like parsers and type checkers for analysis of data according to the specifications. Polyglot [10] extracts protocol message formats by dynamic analysis of binaries. Static analysis has been used to extract output formats from stripped executables [23] and data models from programs [21].

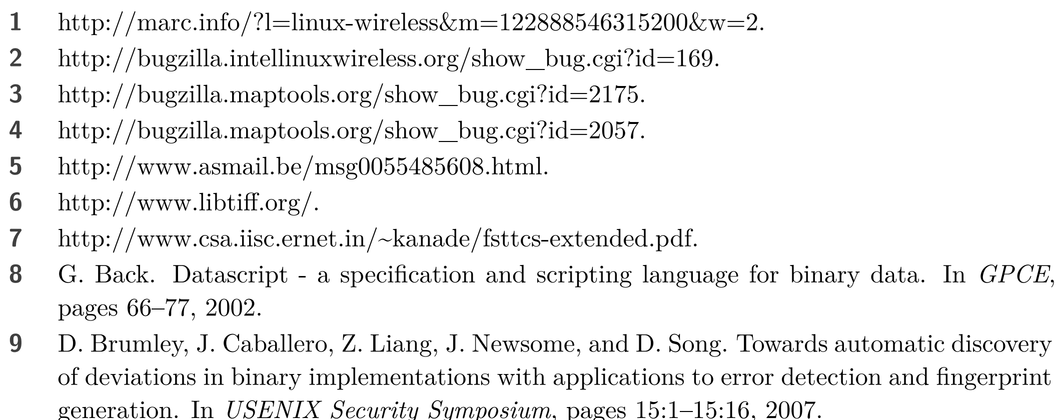
Type inference. Physical type checking [12] computes the in-memory layout of C structures to prove type safety of field accesses in presence of pointer type-casts. Our layout maps are defined over an ordered set of offset-based memory locations and are associated with guards. Type inferencing approaches [30, 22] have been used for extracting data abstractions from Cobol programs. Our idea of inferring concrete types based on accesses rather than on type declarations is similar to the use of access patterns in a specific program for identification of aggregate structures [30]. Our offset-based uniform naming is similar to their atomization notion. Our analysis is however aimed at C programs and we associate guards with layout maps. There are several other type systems, like dependent types for imperative languages [34], liquid types [31], and predicated type hierarchy [19], which associate guards with types.

7 Conclusions and Future Work

We presented an approach to check data format compatibility of C programs. Our approach infers data formats by static analysis of programs and checks various notions of compatibility of producer-consumer and version-related programs. It was shown to be effective in finding bugs in real programs. We plan to extend the approach to unbounded-length data and explore its use in improving regression testing on data formats. The integration of guarded layouts with finite state machine specifications of protocols can help in checking richer notions of compatibility of communicating programs.

Acknowledgments. We thank Satish Chandra and Nishant Sinha for their valuable feedback on the earlier versions of this paper.

References



1 http://marc.info/?l=linux-wireless&m=122888546315200&w=2.
2 http://bugzilla.intellinuxwireless.org/show_bug.cgi?id=169.
3 http://bugzilla.maptools.org/show_bug.cgi?id=2175.
4 http://bugzilla.maptools.org/show_bug.cgi?id=2057.
5 http://www.asmail.be/msg0055485608.html.
6 http://www.libtiff.org/.
7 http://www.csa.iisc.ernet.in/~kanade/fsttcs-extended.pdf.
8 G. Back. Datascript - a specification and scripting language for binary data. In *GPCE*, pages 66–77, 2002.
9 D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *USENIX Security Symposium*, pages 15:1–15:16, 2007.

10 J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *CCS*, pages 317–329, 2007.
11 S. Chaki, E. M. Clarke, N. Sharygina, and N. Sinha. Verification of evolving software via component substitutability analysis. *FMSD*, 32(3):235–266, 2008.
12 S. Chandra and T. Reps. Physical type checking for C. In *PASTE*, pages 66–75, 1999.
13 P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
14 L. de Alfaro and T. Henzinger. Interface automata. In *FSE*, pages 109–120, 2001.
15 E. Driscoll, A. Burton, and T. Reps. Checking conformance of a producer and a consumer. In *FSE*, pages 113–123, 2011.
16 M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*, pages 242–256, 1994.
17 M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J.R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*, pages 177–190, 2006.
18 K. Fisher and D. Walker. The PADS project: an overview. In *ICDT*, pages 11–17, 2011.
19 R. Jhala, R. Majumdar, and R. Xu. State of the union: Type inference via Craig interpolation. In *TACAS*, pages 553–567, 2007.
20 A. Kanade, R. Alur, S. Rajamani, and G. Ramalingam. Representation dependence testing using program inversion. In *FSE*, pages 277–286, 2010.
21 R. Komondoor and G. Ramalingam. Recovering data models via guarded dependences. In *WCRE*, pages 110–119, 2007.
22 R. Komondoor, G. Ramalingam, S. Chandra, and J. Field. Dependent types for program understanding. In *TACAS*, pages 157–173, 2005.
23 J. Lim, T. Reps, and B. Liblit. Extracting output formats from executables. In *WCRE*, pages 167–178, 2006.
24 B. Liskov and J. Wing. Behavioral subtyping using invariants and constraints. In *Formal Methods for Distributed Processing: An Object Oriented Approach*, pages 254–280. 2001.
25 L. Mariani, S. Papagiannakis, and M. Pezzè. Compatibility and regression testing of COTS-component-based software. In *ICSE*, pages 85–95, 2007.
26 S. McCamant and M.D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *ECOOP*, pages 440–464, 2004.
27 P.J. McCann and S. Chandra. Packet types: Abstract specifications of network protocol messages. In *SIGCOMM*, pages 321–333, 2000.
28 A. Miné. The octagon abstract domain. *CoRR*, abs/cs/0703084, 2007.
29 G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC*, pages 213–228, 2002.
30 G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *POPL*, pages 119–132, 1999.
31 P. M. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL*, pages 131–144, 2010.
32 N. Sharygina, S. Chaki, E.M. Clarke, and N. Sinha. Dynamic component substitutability analysis. In *FM*, pages 512–528, 2005.
33 Z. Stengel and T. Bultan. Analyzing singularity channel contracts. In *ISSTA*, pages 13–24, 2009.
34 H. Xi. Imperative programming with dependent types. In *LICS*, pages 375–387, 2000.
35 K. Q. Zhu, K. Fisher, and D. Walker. Learnpads++: Incremental inference of ad hoc data formats. In *PADL*, pages 168–182, 2012.