SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **40**:225–238 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Serializing C intermediate representations for efficient and portable parsing (preprint)

Jeffrey A. Meister[1], Jeffrey S. Foster[2,*], and Michael Hicks[2]

[1] *Department of Computer Science and Engineering, University of California, San Diego, CA 92093, USA*
[2] *Department of Computer Science, University of Maryland, College Park, MD 20742, USA*

SUMMARY

C static analysis tools often use intermediate representations (IRs) that organize program data in a simple, well-structured manner. However, the C parsers that create IRs are slow, and because they are difficult to write, only a few implementations exist, limiting the languages in which a C static analysis can be written. To solve these problems, we investigate two language-independent, on-disk representations of C IRs: one using XML, and the other using an Internet standard binary encoding called XDR. We benchmark the parsing speeds of both options, finding the XML to be about a factor of two slower than parsing C and the XDR over six times faster. Furthermore, we show that the XML files are far too large at 19 times the size of C source code, while XDR is only 2.2 times the C size. We also demonstrate the portability of our XDR system by presenting a C source code querying tool in Ruby. Our solution and the insights we gained from building it will be useful to analysis authors and other clients of C IRs. We have made our software freely available for download at `http://www.cs.umd.edu/projects/PL/scil/`.

KEY WORDS: C, static analysis, intermediate representations, parsing, XML, XDR

1. Introduction

There is significant interest in writing static analysis tools for C programs. The *front end* of a compiler or static analyzer parses a program's C source code to yield a conveniently structured intermediate representation (IR), which is the object of subsequent analysis. Typical IRs include abstract syntax trees (ASTs) and control-flow graphs (CFGs).

In most front-ends, IRs are created in-memory and are discarded when the client analysis completes. This has several unfortunate consequences. For one, each separate analysis program

[*]Correspondence to: Jeffrey S. Foster, 4129 A.V. Williams Bldg #115, Department of Computer Science, University of Maryland, College Park, MD 20742, USA, jfoster@cs.umd.edu

Copyright © 2010 John Wiley & Sons, Ltd.

Received 13 June 2008
Revised 5 May 2009
Accepted 6 November 2009

must reapply the front-end to parse the source code. Parsing is time consuming, and the cost it adds to simple analyses may be significant. Worse, analyses are most easily written in the same language as the front-end, forcing developers to know or learn that language even if it is not well-suited to their application. Since it is very difficult to create a C front-end, analysis writers are effectively restricted to coding their analyses in the small subset of programming languages in which high-quality front-ends already exist.

In this paper, we investigate language-independent, on-disk representations of IRs for the C programming language. Our aim is to develop a simple representation that is fast and easy to parse. If parsing is easy, it will be straightforward to write parsers for those languages in which a full C source-language parser does not exist (e.g., Ruby, Perl, Python, Haskell, etc.). If parsing is fast, repeatedly parsing a large program, e.g., to run different analyses, will have lower overhead.

The starting point of our approach is CIL, a popular and robust C front-end with a well-designed IR [15]. CIL's IR is simpler than C's abstract syntax tree, as it translates many difficult or redundant source-language constructs into simpler ones, simplifying subsequent analysis. CIL is written in OCaml, and thus its use is currently restricted to developers who know that language. We developed two portable serialization formats for this IR: one using XML [25], a well-known general-purpose markup language, and another using the eXternal Data Representation (XDR) [19], an Internet standard binary encoding. We call these implementations XML-C and XDR-C, respectively.

We measured the parsing times of IRs encoded in each of our serialization formats against two baselines: CIL parsing C source code and the OCaml standard library's marshaller decoding CIL IRs serialized in OCaml's custom binary format. The OCaml marshaller's format is heavily dependent on OCaml internals and thus is not easily portable, but it gives a reasonable lower bound on deserialization time in OCaml. Our benchmark suite is composed of 21 open-source C applications totaling 843,850 lines of code. The benchmarking results show that XML-C is slow and bloated—slower than parsing C by about a factor of two on average, with on-disk representations roughly 19 times larger. In contrast, XDR-C is fast and compact—it is about 6.7 times faster to parse than C source, while its on-disk representations are only 2.2 times larger. Its speed and space characteristics are close to those of OCaml's native format.

To demonstrate XDR-C's portability we implemented a decoder for XDR-C in the Ruby scripting language. We implemented a simple querying system on top of this decoder and found that it was very easy and straightforward to write. This experience, and our performance measurements, suggest that XDR-C reasonably meets our goal of efficient and portable parsing. We hope the system proves useful to authors of C static analyses and other clients of C intermediate representations. XDR-C can be downloaded at `http://www.cs.umd.edu/projects/PL/scil/`.

```
type stmtkind =
 | Return of exp option * location
 | Goto of stmt ref * location
 | If of exp * block * block * location
 | Loop of block * location * stmt option * stmt option
 | ...
```

Figure 1. A small excerpt from the CIL data type

2. Serializing the CIL Intermediate Representation

To serialize a C IR to disk, we first have to obtain such an IR from an existing C parser. We chose to use CIL [15], a well-tested and widely-used[†] C front-end written in Objective Caml (or OCaml, for short).[‡] CIL's IR is particularly easy to use because it is a simplified subset of C, with most of C's ugly corners removed.

Figure 1 shows a small excerpt from the type for statements in CIL's IR. Statements are represented by members of the tagged union type `stmtkind`, four cases of which we list for illustration: `Return` statements, which optionally contain the returned expression (represented by a value of type `exp`); `Goto` statements, containing a pointer to the target statement; `If` statements, containing the guard and the true and false branches; and `Loop` statements, which are generic loops containing the loop body and optional break and continue statements. Each `stmtkind` variant also contains a `location` indicating where the statement appeared textually in the original program.

The principal challenge in serializing the CIL IR is deciding how to represent its components, including primitive types (integers, floating-point numbers, strings, booleans, etc.), tagged unions (as above), records, tuples, and lists. A suitable serialization format must directly implement all of these without flattening the structure of the IR, and must be parseable both quickly and portably. We should also note that, although the CIL IR is similar to an AST, it is not a true tree. For example, `Goto` statements include a pointer to their target, which may be an arbitrary node in the IR. Thus our serialization format must also support cross-references.

2.1. Serialization using XML

We initially chose XML as a serialization format. Due to its popularity, almost every modern programming language can parse XML data to a sensible in-memory representation, and thus XML is highly portable. Moreover, there are many existing tools to help programmers

[†]For example, CIL is used by CCured [14], Locksmith [17], and Deputy [3], to name but three mature tools.
[‡]OCaml is a general-purpose, mostly functional programming language with added object-oriented features, "designed with program safety and reliability in mind" [2].

```
1  <statement>
2    <instruction kind="functionCall">
3      <functionCall>
4        <name>
5          <expression kind="lvalue">
6            <lvalue>
7              <base kind="variable">
8                <variableUse name="printf" id="264" isGlobal="true"
9                    isAddressTaken="false" isUsed="true"/>
10             </base>
11           </lvalue>
12         </expression>
13       </name>
14       <argument>
15         <expression kind="constant">
16           <constant kind="string">
17             <value>hello, world&#xA;</value>
18           </constant>
19         </expression>
20       </argument>
21     </functionCall>
22     <location file="hello.c" line="4" byte="7870"/>
23   </instruction>
24 </statement>
```

Figure 2. XML serialization of `printf("hello, world\n")`

manipulate XML. For example, XQuery implementations provide flexible database-like query facilities to extract information from XML trees. XML also seemed likely to take less time to parse than C source code, since C parsing is complex (it is not even quite LALR), whereas XML was designed with simplicity and ease of parsing in mind. Thus, it appeared XML could handily meet our goals of portability and efficiency.

To evaluate XML's potential, we developed a plug-in to serialize the CIL IR to XML. Figure 2 shows the serialization of the statement `printf("hello, world\n")`. The format of the tree is essentially a direct mapping from the CIL IR. Primitive types are represented in the obvious textual manner, e.g., the string "`hello, world`" on line 17 is written out as-is. Records, tuples, and lists are mapped to sequences of XML elements with a common parent. For example, the call to the function `printf` is represented in the CIL IR as a tuple containing the called function and its argument. This tuple is stored in XML by encoding the expression describing the called function (lines 4 through 13) immediately followed by the argument (lines 14 through 20), with both contained as children inside the element `<functionCall>`. Discriminated unions are mapped to an element that names the union and has an attribute encoding the discriminant, and then the values in that arm of the union are stored as children. For example, line 15 shows an instance of the expression union type that is a constant, and the encoding of the constant follows on lines 16–18. Simple property relationships among parts of the IR translate to attributes inside an appropriate element. For example, `printf` has global scope, and this is indicated with a Boolean attribute at the end of line 8.

The only minor subtlety is back- and cross-references in the IR. We assign a unique ID number to each node when it first appears, and then we use the ID if we need to refer back to that node. For example, `printf` is referred to by its ID (264) on line 8.

One potential problem with the XML IR in Figure 2 is immediately obvious: it is much larger than the equivalent C source code. When working with IRs, some size increase is expected and is perhaps unavoidable. One of the reasons that IRs are much easier to analyze than concrete program syntax is that they make explicit a lot of structural information that is implicit in the source code. For instance, the descending chain of XML elements on lines 1-8 is not explicit in the original source code; the programmer garners it from his or her knowledge of the C language and the context in which `printf` appears.

However, not all of the example's verbosity can be ascribed to the nature of IRs. Some of it is due to the XML format itself. In fact, two aspects of the format's design run contrary to our needs. First, terseness of XML markup was considered unimportant by the creators of XML [29]. We, on the other hand, require an efficient representation if we hope to achieve a reasonable increase in parsing speed. Second, XML was designed to be readable by humans. This is not useful to us, since we do not intend for our serialized IRs to be directly edited (surely, source code is better for this purpose).

The negative consequences of the XML design are quite apparent from Figure 2. Every element is given a meaningful name, even though such labels are unnecessary for unique identification. Worse, this name must be repeated in its entirety to close any non-leaf element; this extravagance may improve human readability, but it is not required by parsers, for which a single closing character would suffice. The XML syntax contains many meta-characters, including angle brackets, quotation marks, equal signs, and slashes, which serve to group tokens of the XML document in a legible fashion but do not themselves encode any data.

XML's bloated nature results in parse times and file sizes that are too inefficient for general use, as we show in Section 3. We therefore turned our attention to an alternative format, XDR.

2.2. Serialization using XDR

XDR is a binary data description and encoding standard introduced by Sun Microsystems in the late 1980s. The XDR standard [21] includes two key components: a language used for defining data structures and a binary format for representing those structures. An XDR implementation for a given programming language takes a definition of a data structure, written as an XDR data description, and generates functions to convert between the programming language's native representations of data and the XDR binary format.

XDR is the intermediate encoding used for passing data in Sun's remote procedure call (RPC) implementation. Sun RPC libraries are widely available, and hence code to handle XDR is broadly available. Thus, although XDR does not enjoy the popularity of XML, libraries that support it are available in many programming languages.

Figure 3 shows the XDR encoding of the CIL representation of `printf("hello, world\n")`. The first column numbers each byte, the second column displays each 32-bit word of the

Byte Index	Hex Word	Comment
0	`00 00 00 00`	The statement has no labels.
4	`00 00 00 00`	It is a sequence of instructions
8	`00 00 00 01`	with only one member:
12	`00 00 00 01`	a function call.
16	`00 00 00 00`	The result is discarded.
20	`00 00 00 01`	The function name is an lvalue,
24	`00 00 00 00`	with a host consisting of a variable
28	`00 00 01 08`	whose ID is 264
32	`00 00 00 00`	and no offset.
36	`00 00 00 01`	There is one argument:
40	`00 00 00 00`	a constant,
44	`00 00 00 01`	in particular, a string,
48	`00 00 00 0d`	which is 13 characters long:
52	`68 65 6c 6c`	`h e l l`
56	`6f 2c 20 77`	`o , w`
60	`6f 72 6c 64`	`o r l d`
64	`0a 00 00 00`	`\n`
68	`00 00 00 04`	The statement occurs at line 4
72	`00 00 00 07`	of the file with the 7-character name
76	`68 65 6c 6c`	`h e l l`
80	`6f 2e 63 00`	`o . c`
84	`00 00 1e be`	starting at byte 7870.
88	`ff ff ff ff`	No CFG information was generated,
92	`00 00 00 00`	so there are no known successors
96	`00 00 00 00`	or predecessors.

Figure 3. XDR serialization of `printf("hello, world\n")`

encoding (XDR-encoded values are 4-byte aligned), and the third column explains what each word means. XDR specifies the expected C-like encodings for primitive types: `signed` and `unsigned int`s are 32 bits long and in big-endian byte order, there is a type for 64-bit integers called `hyper`, `float` and `double` follow the IEEE standard, and so on. For example, the word at byte 28 encodes the integer 264. We represent lists with XDR arrays, which are encoded as an `unsigned int` member count followed by the elements of the array in succession. For example, the list of arguments to the function `printf` begins at byte 36, where the 32-bit integer 1 is encoded to indicate that there is just a single argument, whose encoding then follows until byte 67. Strings are similarly encoded with a length followed by ASCII data, but the bytes used to encode the string are zero-padded to a multiple of 4, as shown at bytes 65–67 and 83. Records and tuples are stored using XDR `struct`s, which are encoded as simply each of their elements concatenated together. For example, an lvalue is made up of a host and an offset, and the lvalue beginning at byte 20 encodes the host at bytes 24–31 and the offset at bytes 32–35. Discriminated unions are directly supported as XDR `union`s, encoded as an integral type identifying the arm that is present followed by that arm. For example, the 32-bit integer 1 at byte 44 signifies that the constant encoded there is a string constant, and the string follows thereafter.

It is immediately evident that for this example, the XDR IR is shorter (100 bytes) than the XML IR (462 bytes, ignoring indentation and newlines), though they encode the same CIL data structure. For instance, compare lines 14–20 of the XML with bytes 40–67 of the XDR. The XML's readable tags clearly explain that the constant string `hello, world\n` is the argument to the function `printf`. In contrast, it is not easy to discern the meaning of the XDR-encoded information without the explanatory comments. The word at byte 40 identifies some of the following data as a constant expression, but it is not clear how `00 00 00 00` conveys that information or where the constant ends. An XDR decoder uses a data description to determine the meanings of values, and unlike an XML parser, an XDR decoder cannot extract any meta-data from the values without the data description. For this reason, XDR is said to be an *implicitly typed* encoding [21].

An XDR data description assigns type information to encoded data by building up composite types such as structures, enumerations, and discriminated unions, starting from primitive types. XDR type definitions look very similar to C type definitions. For instance, here is the type definition of a CIL statement:

```
struct stmt {
  label stmt_labels<>;
  real_stmt stmt_content;
  int stmt_cfg_id;
  stmt stmt_cfg_successors<>;
  stmt stmt_cfg_predecessors<>;
};
```

Comparing Figure 3 with this data description, we see that a `struct` in XDR is encoded as each of its items concatenated together, so each statement begins with an array of `labels` (the `<>` syntax signifies a variable-length array). Since an array is encoded as an `unsigned int` length followed by each member, and since there are no labels in our example, the first 4 bytes of the XDR encoding are `00 00 00 00`. The next item in the `struct` is a `real_stmt`, which is another user-defined type. Its definition is not shown, but it is a discriminated union that uses `ints` 0 to 10 to identify which kind of statement is present. Possibilities include `return` statements, `gotos`, `breaks`, and so on. The `int` 0 refers to the kind of statement that is just an array of instructions, which is what we have in our example. Thus, the next 4 bytes are `00 00 00 00`, followed by `00 00 00 01` to indicate an array length of 1, and then encoding of an instruction begins. The instruction ends at byte 87; afterward, the last three items of the `stmt` appear, and the encoding ends.

As we did with XML, we constructed an XDR-encoding CIL plug-in by hand. We wrote the plug-in following a straightforward, mechanical translation of the data description into OCaml code. We experimented with a toolkit that can build an OCaml XDR parser automatically from a data description file, part of the Ocamlnet package's implementation of Sun RPC [16]. Unfortunately, we found that this generator was unable to handle our large and complex data description [24], though we did make use of its mapping between OCaml primitive types and XDR primitive types.

One possible problem with XDR is that, since the smallest encodable unit is four bytes long and all values must be padded to a length that is a multiple of four, some bytes of the encoded file do not store any data. For example, there are many discriminated unions in the CIL IR, and each time a discriminant is encoded, four bytes must be used even though one would suffice. However, the four-byte unit size of XDR was an explicit design choice. The creators of the format wanted to avoid causing alignment problems on as many machines as possible without sacrificing too much space, so they chose four as a compromise [20].

Compared to XML, the XDR format is clearly more compact. There are no meta-characters, since XDR is not a text format and was never intended to be human-readable. No complicated parsing techniques are necessary: since the length of every item is always known before that item is encountered, the decoding process is merely a forward procession through a byte stream. Indeed, since the XDR files are small and simple, the decoding process is much faster than parsing C source code, fulfilling our speed goal, as we show in Section 3. Moreover, since XDR is a portable format our system enables C analyses to be written in a wide variety of programming languages, as we illustrate in Section 4.

Note that a possible alternative to XDR is to use a binary encoding of XML, such as Fast Infoset [27], Efficient Binary Meta Language (EBML) [5], or the encoding advocated by the Binary Characterization Working Group [26]. Like XDR, Binary XML is intended to be terse, to reduce parse times and storage requirements. However, there is no single, widely supported standard—as a primary motivation behind using XML is its portability, Binary XML is not a suitable substitute. By contrast, XDR is both standardized and widely supported.

3. Experimental Results

We evaluated the efficiency of our serialization formats on a suite of 21 open-source C applications totaling 843,850 lines. We were most interested in two measures: the size of a program in the encoded format as compared to the C original, and the time to parse a program in the encoded format, as compared to parsing the source code from scratch. Our benchmarking machine has a dual-core 2.80 GHz Intel Xeon CPU with 4 GB RAM and runs Linux kernel version 2.6.9-67.

We used the following methodology. First, we parsed each program file using CIL and serialized it in three formats: XML-C, XDR-C, and a direct encoding of the CIL IR using OCaml's `Marshal` module, which performs OCaml-specific object serialization and deserialization. Though OCaml `Marshal` is not a viable serialization format for our purposes, since it is OCaml-specific and subject to change with each new OCaml revision, it is highly optimized and thus bounds how compact and fast deserialization (and serialization) can be. We place XML-C at less of a disadvantage by restricting all element and attribute names to four characters, yielding a slightly more compact representation than Figure 2.

Next, we measured the size of each encoded file and the time to deserialize it, comparing the results to the size and time to parse the original C code using CIL. We find that XDR-C is quite fast—several times faster than parsing C code directly—while XML-C, no matter the implementation of the XML parser, either provides no speed improvement or degrades performance. We perform further experiments to understand the effects of serialized file size

Program	Lines of	MBytes of			
	Code	C Source	XML-C	XDR-C	`Marshal`
bind-9.3.4	222,176	13.532	195.931	23.658	12.048
gettext-0.16	102,909	13.628	137.616	15.121	7.623
zebra-0.95	149,662	5.530	84.048	10.870	5.562
openssh-4.2p1	53,252	3.063	44.641	6.229	3.185
gawk-3.1.5	35,623	2.120	43.213	4.934	2.273
apache_1.3.1	48,251	1.874	31.316	3.961	1.927
bison-2.3	53,954	1.451	23.465	3.228	1.519
wget-1.9	23,317	1.140	22.154	2.918	1.389
xinetd-2.3.14	16,260	0.965	14.218	1.944	0.983
retawq-0.2.6c	22,881	1.275	23.249	2.775	1.239
less-382	15,553	0.725	12.752	1.968	0.871
gnuchess-5.07	9,245	0.726	15.617	2.012	0.965
make-3.81	24,062	0.891	17.062	1.821	0.873
nano-2.0.3	14,202	0.716	12.023	1.485	0.714
vsftpd-2.0.3	11,743	0.666	9.695	1.058	0.559
sed-4.1	21,641	0.721	13.735	1.486	0.736
bc-1.06	8,510	0.431	7.719	1.168	0.538
gzip-1.24	5,809	0.232	4.438	0.528	0.252
which-2.16	2,099	0.065	1.174	0.134	0.064
time-1.7	1,395	0.034	0.536	0.084	0.036
spell-1.0	1,306	0.041	0.629	0.090	0.047
Total || Arith. mean	843,850	2.373	34.059	4.165	2.067

Table I. Line counts and total file sizes for each format (in megabytes)

and disk access latencies on performance, and confirm that the source of XML-C's poor performance is the verbosity of the XML encoding.

Size of serialized representations. Table I lists each benchmark along with its line count, as determined by SLOCCount [23], and its total size in megabytes in original source form and in each of three serialization formats. Figure 4 graphs these results, showing the ratios of the serialization format sizes with the C source size. The last column in Figure 4 gives the geometric mean of the ratios.§

We can see that while a typical XDR-C representation is (as expected) larger than the C source code, it remains within a small constant factor. On average, the XDR-C is 2.2 times the size of C. OCaml's native format, since it can take advantage of the OCaml representation of values in memory to optimize for size, manages to be about the same size as C source code (with a ratio of 1.024) despite storing the more verbose structure of the IR. Unlike XDR-C, OCaml's format does not require values to be padded to a multiple of four bytes, so it saves

§The geometric mean is the appropriate single-point summarizer because the values are ratios without units [11]. Note that our use of stacked bars in Figure 4 aims to save space (i.e., rather than graphing three side-by-side bars per program); we do not mean to imply that the cost of one encoding adds to the costs of the others.

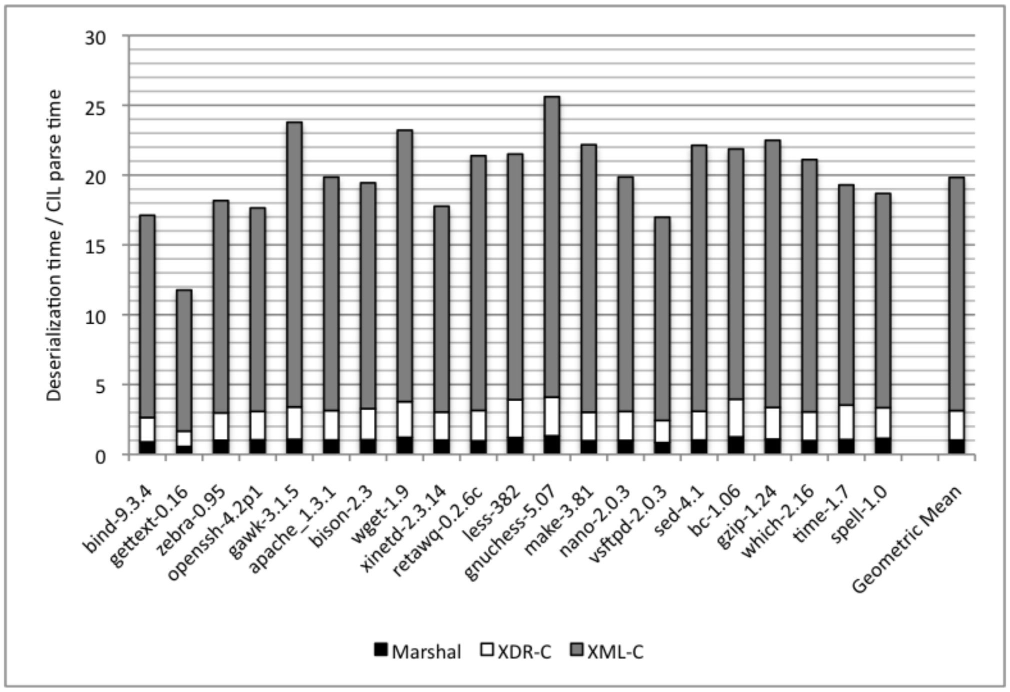

Figure 4. Total file sizes for each format compared to the equivalent C source code sizes

some space. XML-C, however, is much larger than C, sometimes ludicrously so: the XML-C for bind-9.3.4 takes up almost 200 MB. On average, XML-C is 19 times the size of C. As we will see next, this dramatic size difference has a strong affect on deserialization performance.

Deserialization times. The left portion of Table II lists, for each benchmark, the time to parse the original source with CIL, as a baseline, and the times to deserialize the XDR-C format using our custom code; to deserialize the OCaml `Marshal` representation; and to deserialize the XML-C format into OCaml using PXP [18]. Each of the parsers listed in columns 2–5 is written in OCaml, and we compiled each to native code using OCaml compiler version 3.10.0. The right portion of the table lists deserialization times for XML-C with other XML parsers, but we defer discussion of these for the moment. Each reported time is the median of 11 runs, after one discarded run to warm up the buffer cache.

Figure 5 graphs these results, showing the ratios of the each of the XML-C with PXP, XDR-C, and OCaml Marshal deserialization times to the CIL parsing time. The figure shows that parsing XML-C with PXP is roughly twice as slow as directly parsing the C source with CIL. XDR-C, on the other hand, is significantly faster to parse than the equivalent source, on average taking 0.15 times as long. OCaml's `Marshal` deserialization is even faster, taking

Program	CIL	XDR-C	Marshal	XML-C PXP	XML-C with Xerces	SCEW	Expat
bind-9.3.4	19.425	3.108	0.817	54.918	31.146	16.366	4.761
gettext-0.16	12.150	2.075	0.668	26.310	16.792	12.659	2.380
zebra-0.95	15.674	1.491	0.425	24.934	14.574	7.340	2.206
openssh-4.2p1	24.163	1.372	0.459	14.903	10.385	4.193	1.375
gawk-3.1.5	3.667	0.552	0.125	11.338	5.993	3.474	0.956
apache_1.3.1	5.727	0.610	0.188	9.296	5.585	2.695	0.816
bison-2.3	3.893	0.589	0.174	7.682	4.914	2.226	0.681
wget-1.9	2.426	0.408	0.113	6.674	3.789	2.005	0.562
xinetd-2.3.14	3.773	0.489	0.177	4.606	3.463	1.297	0.442
retawq-0.2.6c	1.272	0.273	0.046	6.061	3.076	1.873	0.507
less-382	2.357	0.348	0.105	4.559	2.798	3.554	0.387
gnuchess-5.07	2.222	0.362	0.103	4.764	2.883	1.349	0.409
make-3.81	2.618	0.283	0.089	4.420	2.658	1.342	0.397
nano-2.0.3	1.919	0.208	0.060	3.333	1.961	1.005	0.296
vsftpd-2.0.3	1.138	0.273	0.102	2.700	1.989	0.781	0.262
sed-4.1	1.035	0.185	0.048	3.555	1.922	1.091	0.304
bc-1.06	1.002	0.208	0.065	2.629	1.612	0.768	0.225
gzip-1.2.4	0.643	0.121	0.046	1.377	0.942	0.392	0.128
which-2.16	0.199	0.046	0.020	0.376	0.294	0.104	0.038
time-1.7	0.200	0.042	0.019	0.230	0.225	0.057	0.026
spell-1.0	0.159	0.031	0.014	0.243	0.193	0.065	0.026
Arithmetic Mean	5.032	0.623	0.184	9.281	5.581	3.078	0.818

Table II. Parsing times, in seconds, of each data representation format (median of 11 trials)

on average only 0.048 times as long as parsing C, giving probably the best possible (but platform-dependent) performance.

Based on the benchmark data, we observe that XML-C takes on average only twice the C parsing time despite being 19 times larger. In fact, PXP parses XML at roughly 11,078k bytes per second. This is faster than CIL parses C, roughly 1,500k bytes per second. Unfortunately, when using XML to store IRs, the massive size of the files overwhelms any speed benefit gained by using a simpler format.

Similar reasoning can be applied to XDR-C but with the opposite conclusion. Our XDR-C decoder processes about 13,747k bytes per second. This rapid parsing speed, combined with the reasonably small size of the XDR-C files, makes IRs stored in XDR faster to parse than the equivalent C source code. Thus, we can conclude from these results that XDR-C best meets our speed goals.

XML deserialization with other libraries. We used PXP in our experiments because it gives the most fair comparison: it is written in OCaml, like our other parsers and decoders, and it builds an in-memory tree representation of the XML-C document. A client analysis could use this data structure in lieu of the CIL IR. However, we should point out that this data structure is likely inconvenient to work with and may require further transformation—and

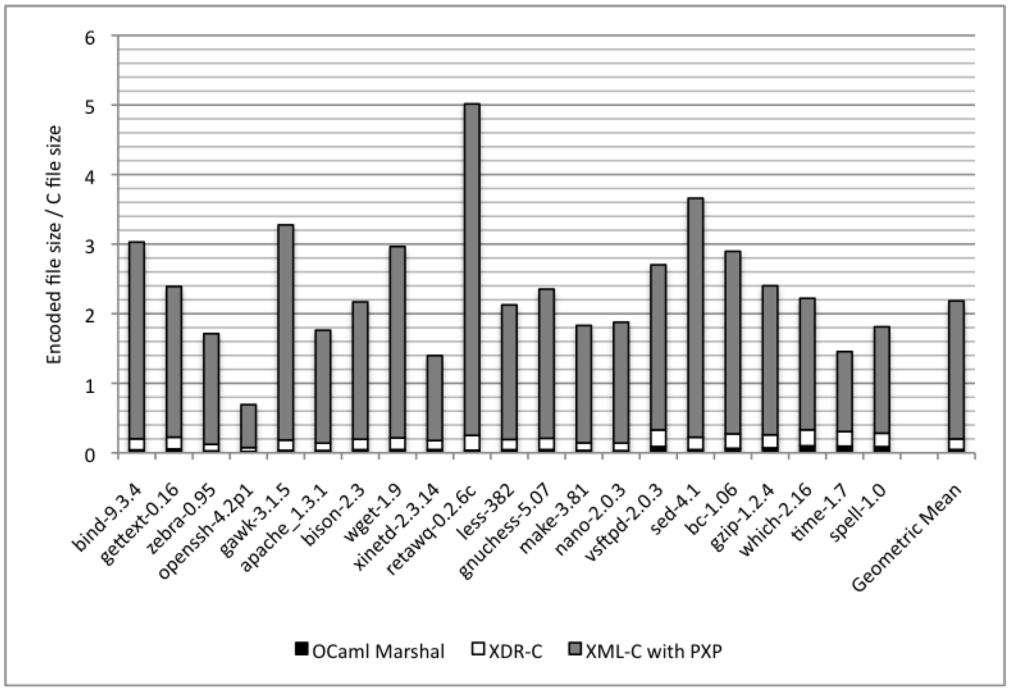

Figure 5. Deserialization times of each data representation format compared to the CIL parsing time

thus the times we report for PXP and other XML parsers are effectively lower bounds on the times that would be required in practice.

However, there are other, faster XML parsers available that are worth considering. The last three columns of Table II give the times for several parsers we benchmarked. Xerces [28] is a portable C++ XML parser with full DOM support developed by the Apache project. It is faster than PXP, taking 64% as long for parsing on average, but it is still slower than parsing with CIL. The Simple C Expat Wrapper (SCEW) [22] is faster than the CIL parser, but still not as fast as XDR-C. SCEW is built on top of Expat [6], which is a SAX parser, meaning it invokes programmer-provided callbacks during parsing. In SCEW, these callbacks are used to construct an in-memory representation of the XML tree. The rightmost column in Table II gives the deserialization times if we run Expat with empty callbacks, which takes approximately 26% of the SCEW time, suggesting that building the in-memory representation takes the majority of the time. Even without this critical step (since we expect most clients of this system to want an in-memory representation), Expat is still slower than XDR-C.

Considering the effects of I/O. A potential issue with the experimental results presented so far is that, since the files in our benchmark suite vary widely in size, disk access times could

Program	C (CIL)	XDR-C	Marshal	XML-C with			
				PXP	Xerces	SCEW	Expat
bind-9.3.4	0.910	0.986	0.856	0.918	1.048	1.002	1.004
gettext-0.16	0.912	1.003	0.819	0.910	1.024	1.006	1.002
zebra-0.95	0.914	0.992	0.828	0.916	1.002	0.998	1.003
openssh-4.2p1	0.900	0.993	0.837	0.917	1.005	0.999	1.000
gawk-3.1.5	0.848	1.002	0.840	0.911	1.027	1.003	1.008
apache_1.3.1	0.907	1.007	0.814	0.897	1.061	0.995	1.002
bison-2.3	0.905	1.007	0.862	0.905	1.009	0.998	1.006
wget-1.9	0.902	1.007	0.832	0.910	1.001	0.990	1.005
xinetd-2.3.14	0.911	0.994	0.797	0.914	1.006	0.995	1.005
retawq-0.2.6c	0.895	0.963	0.935	0.924	0.998	0.985	1.002
less-382	0.859	1.009	0.829	0.916	1.005	1.001	1.003
gnuchess-5.07	0.899	0.975	0.845	0.906	1.004	1.000	1.002
make-3.81	0.874	0.993	0.798	0.917	1.000	1.007	1.005
nano-2.0.3	0.913	0.995	0.817	0.911	0.997	1.000	1.010
vsftpd-2.0.3	0.895	1.011	0.794	0.912	1.005	0.992	1.008
sed-4.1	0.880	0.957	0.833	0.915	1.006	0.984	1.010
bc-1.06	0.906	1.014	0.800	0.892	1.017	0.995	1.009
gzip-1.2.4	0.813	1.041	0.783	0.904	1.005	1.000	1.008
which-2.16	0.910	1.000	0.800	0.896	1.010	0.990	1.026
time-1.7	0.880	1.000	0.789	0.900	1.000	1.018	1.000
spell-1.0	0.868	1.000	0.857	0.901	1.005	1.000	1.000
Geometric Mean	0.890	0.997	0.826	0.909	1.011	0.998	1.006

Table III. Parsing improvement due to RAM disk (RAM disk parse time / HD parse time)

be a significant contributing factor. Indeed, because XML files are generally much larger than XDR files, it is possible that the greater slowdown for XML is in large part due to disk access. To account for this possibility, we performed two additional sets of measurements. First, we performed the same set of experiments, but with each benchmark stored on and read from a 250 MB RAM disk. (Although this takes away from the amount of RAM available to our parsers, we did not observe any paging, so it is unlikely that they were affected.) Second, we compressed the benchmark files to reduce their on-disk size, and then performed our experiments while decompressing these files on-the-fly, during parsing. We find that neither the RAM disk nor on-the-fly decompression are effective at consistently reducing parsing times.

Table III presents the results of this repeated experiment as a ratio of RAM disk times to hard disk times, indicating how much the use of a RAM disk improved our parsing times. The results are varied: CIL and PXP parse in 0.890x and 0.909x their original times, while our XDR decoder is virtually unaffected, and OCaml's Marshal library experiences a larger 0.826x improvement. The size of a particular benchmark does not seem to be correlated with its improvement ratio. We suspect that unpredictable variance in disk seek times for different files is making it difficult to compare overheads. In any case, the speed increases from using a RAM disk are small.

In lieu of using a RAM disk, we could try to reduce I/O costs at the cost of increased computation by using on-the-fly (OTF) decompression. Our C source code and XML-C files

Program	C (CIL)	XDR-C	Marshal	XML-C with			
				PXP	Xerces	SCEW	Expat
bind-9.3.4	0.941	1.474	1.618	0.953	1.057	1.099	1.304
gettext-0.16	0.953	1.719	1.557	0.952	1.067	1.083	1.352
zebra-0.95	0.937	1.499	1.562	0.949	1.056	1.099	1.291
openssh-4.2p1	0.923	1.782	1.608	0.958	1.064	1.140	1.337
gawk-3.1.5	0.865	1.246	1.544	0.942	1.052	1.090	1.311
apache_1.3.1	0.934	1.579	1.548	0.932	1.060	1.109	1.314
bison-2.3	0.940	1.649	1.638	0.945	1.056	1.116	1.307
wget-1.9	0.933	1.466	1.558	0.944	1.053	1.090	1.299
xinetd-2.3.14	0.949	1.834	1.525	0.964	1.068	1.160	1.371
retawq-0.2.6c	0.909	1.059	1.696	0.956	1.054	1.065	1.298
less-382	0.896	1.618	1.543	0.952	1.057	1.029	1.284
gnuchess-5.07	0.936	1.539	1.553	0.942	1.058	1.113	1.306
make-3.81	0.901	1.530	1.494	0.960	1.061	1.126	1.332
nano-2.0.3	0.936	1.438	1.533	0.947	1.055	1.101	1.314
vsftpd-2.0.3	0.955	1.813	1.510	0.963	1.070	1.152	1.378
sed-4.1	0.901	1.292	1.521	0.950	1.059	1.082	1.316
bc-1.06	0.951	1.577	1.477	0.930	1.055	1.100	1.298
gzip-1.2.4	0.860	1.752	1.435	0.948	1.068	1.148	1.328
which-2.16	0.970	1.826	1.400	0.952	1.071	1.163	1.368
time-1.7	0.940	1.905	1.421	0.974	1.062	1.263	1.385
spell-1.0	0.925	1.839	1.429	0.955	1.073	1.185	1.308
Geometric Mean	0.926	1.576	1.530	0.951	1.061	1.119	1.323

Table IV. Parsing improvement due to decomp. (OTF gunzip parse time / HD parse time)

are plain text, which compresses well, and the 32-bit alignment of XDR-C introduces many zero bytes that can be easily compressed. We have compressed all our benchmark files with the popular `gzip` [10] tool and repeated our experiments once more, modified so that these compressed files are read from disk and `gunzip`ped in memory before being parsed.

The results, presented in Table IV, show that on-the-fly decompression does not improve parsing times; in fact, sometimes it degrades them. CIL and PXP do consistently show some minor improvement, but it is not enough to be noticeable in practice, even on large benchmarks. Our XDR decoder and OCaml's Marshal, however, are consistently negatively impacted. It is likely that, since the XDR-C and marshalled files are so small, the overhead of having to `gunzip` is visible, and it swamps out any improvement that may have been gained from compression. Note that this slowdown would also generally not be noticed, since XDR and Marshal take less than a second on most benchmarks with or without compression.

Serialization times. While our primary performance concern is deserialization time, and secondarily disk size, as a sanity check we also measured the time to parse our applications with CIL and then serialize them using each encoding format. Taking the median of three runs, we found that encoding XML-C imposes a 180% time overhead on CIL (that is, if it takes X seconds to parse a C source file, it takes an additional $1.8X$ seconds to write it out to disk as XML-C). Encoding XDR-C and OCaml's marshaling format adds just 3.3% and 1.6%, respectively. We conclude that the overhead of using XDR is negligible and close to that of using OCaml's format, while XML files, due to their immense size, take a significant amount of time to write.

4. Case Study: An Analysis Written in Ruby

Having demonstrated that our encoding is efficient, the next step is to show that it can be used to build useful tools, written in a variety of languages. The CIL OCaml implementation is quite mature, and many sophisticated C analysis tools have been written in OCaml that use it, such as CCured [14], Deputy [3], and Locksmith [17], to name just a few. Our work opens the possibility of using the CIL IR in languages other than OCaml. In particular, a program can be parsed using the OCaml CIL parser, serialized to disk using XDR-C, and then subsequently read in by an XDR-C parser written in another language. Because of XDR's regularity and simplicity, we expect this task to be straightforward, especially compared to writing a parser for C source syntax.

We implemented a simple XDR-C parser in the Ruby scripting language, and wrote several simple `ctags`-like analyses [7] that use it. Ruby lacks a generic XDR parsing library, so we built our built our XDR-C parser manually. Fortunately, this task was not difficult—our demonstration program is just under 2,400 lines of code in three Ruby files.

The first step, constructing a mapping from XDR primitive types to Ruby types, is simple in Ruby. The `unpack` method of the `String` class does almost all of the work for us. For example, to decode an XDR `double`, we read 8 bytes of input into a `String` and invoke `unpack` on it with the parameter `"G"`, yielding the corresponding Ruby `Float` object. All the other primitive types are handled in the same way. Similar steps apply to any language that is capable of extracting integers, floating-point numbers, and so on from string data and converting them to the native representation of those types in that language. Arithmetic manipulations may be required in some cases (for instance, if native integers are not stored in big-endian byte order), but these are straightforward.

The second step, building a data structure to hold the IR, should be easy for any skilled programmer. The custom types in XDR-C are built using just three aggregates: arrays, structures, and discriminated unions. Almost all languages have conventional analogs for these forms. In Ruby, we have mapped XDR arrays to Ruby Arrays, XDR structures to Ruby classes (members of the structure become fields of the class), and XDR unions to Ruby inheritance from a common superclass. Once this higher-level mapping is decided upon, the conversion from XDR is mechanical; the data description file shows exactly what each custom type in the IR contains. To complete the decoder, we just need functions (likely recursive ones) to walk through the XDR file, applying the mapping to each type encountered.

Having built a decoder for XDR-C by following this method, we proceeded to add features to show how it might actually prove useful to Ruby programmers. Since we constructed a class hierarchy, we added a variant of the visitor pattern to traverse it. Using that, we implemented visitors to perform a few basic queries on the IR: print all the function calls in a given function, print all the calls to a given function that occur anywhere, find all the global variable uses in

```
class CallsToFuncVisitor < NilVisitor
  def initialize(func_name)
    @func_name = func_name
    @func_calls = []
    @current_func = ""
  end

  def visit_global(global)
    if global.is_a?(AST::FuncDefn)
      @current_func = global.name
    else
      @current_func = ""
    end
  end

  def visit_instr(instr)
    if instr.is_a?(AST::FunCall) && instr.name.identifier == @func_name
      file = instr.loc.file
      line = instr.loc.line
      @func_calls << "#{@current_func} @ #{file}:#{line}"
    end
  end
end
```

Figure 6. A Ruby visitor that finds all calls to a specified function

a given function, and print all the uses of a given global variable that occur anywhere. Similar features are available in `ctags` [7].

Figure 6 gives the complete code for a visitor that finds all the calls to a function. If an instance of this class is created by invoking `CallsToFuncVisitor.new("foo")`, it can be passed to the `accept` method at the root of our Ruby IR. Its field `@func_calls` will then contain the function, file, and line number in which each call to `foo` appears in that program. The code simply keeps track of which function it has most recently entered, and when it encounters an instruction that is a function call to `foo`, it records that instruction's location. The visitor pattern takes care of traversing the IR and ensures that every instruction is visited and checked.

It should be easy to create additional visitors. Essentially, we have built a rudimentary framework for working with C IRs and made it available in Ruby, where no C parser exists or is likely to exist in the foreseeable future.

5. Related Work

Persistent Storage of Parse Data Several applications are similar to ours in that they parse program source code to discover properties about it and then store those properties in files on disk for repeated access later. Saturn [1], a software analysis tool, serializes C IRs as part of its tool chain. It stores relations garnered from parsed C code in syntax databases, and it then runs analyses over these databases, producing summary information and error reports. Saturn also uses CIL as its C front-end, and the information in the syntax databases is stored in OCaml's native (non-portable) marshaling format. The LLVM compiler infrastructure [12] generates bytecode in the LLVM virtual instruction set, a low-level object code representation which it stores on disk. This IR includes type and data flow information, but unlike CIL's IR, it does not maintain a very close relationship with the source program. LLVM is targeted to a different set of program analyses (e.g., compiler optimizations) than CIL's more high-level representation.

The classic source code browsing tools ctags [7] and cscope [4] both parse programs and store some basic data about them in persistent files. Ctags creates an index that locates important language objects in source files, which is typically used to help text editors search for those objects. Cscope also identifies items such as variable and function names and provides querying capabilities similar to our Ruby demonstration program presented in Section 4. Ctags has parsers for 34 programming languages, and cscope can support languages with a C-like syntax, since it uses a single fuzzy parser. Both of these tools have the advantage over CIL that they can handle non-preprocessed source code. However, none of the parsers in ctags or cscope are anywhere near as complex as a mature C parser intended for compilation or analysis, so the data that they store on disk is not as full and accurate as the IRs serialized by our system.

Tools to Simplify Analysis Authoring Several projects have the related goal of making it easier for developers to write custom analysis programs. The Program Query Language (PQL) [13] frees developers from having to work directly with a complicated IR to write their analyses. Instead, PQL provides its own high-level language in which users describe the code patterns that they wish to locate. PQL then automatically generates static and dynamic checkers for these patterns. GEN++ [9] similarly allows developers to build analysis tools using a custom domain-specific language designed for that purpose, and then it generates the actual analysis programs from these specifications. PQL and GEN++ represent an alternative approach to working with the IRs of complex parsing front-ends: automatically generate the actual analysis code to allow developers to work at a higher level.

6. Conclusion

We have investigated on-disk language-independent representations of parsed C code toward the goal of writing efficient and portable C front-ends for analysis tools. We developed two serialization formats for CIL, a well-designed C intermediate representation, one format in XML (called XML-C) and the other in XDR (called XDR-C). We benchmarked the parsing speeds of both formats and found that XDR-C is efficient in time and space, while XML-C was generally inefficient, owing to the large size of XML data. Finally, we built a querying tool for C programs in Ruby using XDR-C. Thus we conclude that XDR-C meets our goals of efficiency and portability.

The insights we gained should be helpful to those exploring the problems of efficient, language-independent serialization of tree-like, structured data. We have made our code available at `http://www.cs.umd.edu/projects/PL/scil/` in the hopes it will be useful for authors of C analysis tools. We believe that our approach can be easily extended handle C variants as well. For example, Frama-C [8] and Deputy [3] both extend C's source syntax to permit stronger forms of verification/safety-checking, and the extensions to XDR-C to serialize/deserialize these annotations would be straightforward.

ACKNOWLEDGEMENTS

This research was funded in part by a gift from Cisco Systems, Inc. Author Jeffrey Meister is currently a graduate student at UC San Diego, but this work was performed primarily while he was an undergraduate at the University of Maryland.

REFERENCES

1. A. Aiken, et al. *An Overview of the Saturn Project.* In *PASTE*, San Diego, CA, USA, 2007.
2. The Caml Language home page. http://caml.inria.fr/ [13 January 2009]
3. J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *Proceedings of European Symposium on Programming (ESOP)*, Braga, Portugal, 2007.
4. Cscope page. http://cscope.sourceforge.net/ [29 April 2008]
5. EBML - Technical Specifications. http://ebml.sourceforge.net/specs/ [5 May 2009]
6. The Expat XML Parser page. http://expat.sourceforge.net/ [7 April 2009]
7. Exuberant Ctags page. http://ctags.sourceforge.net/ [29 April 2008]
8. Frama-C page. http://frama-c.cea.fr/what_is.html [20 January 2009]
9. GEN++ page. http://www.cs.ucdavis.edu/ devanbu/genp/ [29 April 2008]
10. GNU zip page. http://www.gzip.org/ [21 January 2009]
11. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach* (4th edn). Morgan Kaufmann: San Francisco, 2007.
12. The LLVM Compiler Infrastructure Project page. http://llvm.org/ [4 May 2009]
13. M. Martin, B. Livshits, and M.S. Lam. *Finding Application Errors and Security Flaws Using PQL: a Program Query Language.* In *OOPSLA*, San Diego, CA, USA, 2005.
14. G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *TOPLAS*, 27(3):477–526, May 2005.
15. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. *Lecture Notes in Computer Science*, 2304:213–228, 2002.
16. Ocamlnet page. http://projects.camlcity.org/projects/ocamlnet.html [29 April 2008]
17. P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI'06*, pages 320–331, Ottawa, Canada, June 2006.
18. PXP page. http://projects.camlcity.org/projects/pxp.html [29 April 2008]
19. RFC 4506: External Data Representation Standard page. http://tools.ietf.org/html/rfc4506.html [29 April 2008]
20. RFC 4506: External Data Representation Standard page, section 5, item 4. http://tools.ietf.org/html/rfc4506.html [29 April 2008]
21. RFC 4506: External Data Representation Standard page, section 1. http://tools.ietf.org/html/rfc4506.html [29 April 2008]
22. SCEW - Simple C Expat Wrapper page. http://www.nongnu.org/scew/ [7 April 2009]
23. SLOCCount page. http://www.dwheeler.com/sloccount/sloccount.html [29 April 2008]
24. G. Stolpmann, personal correspondence, July 2007.
25. W3C XML page. http://www.w3.org/XML/ [29 April 2008]
26. W3C Binary Characterization. http://www.w3.org/TR/xbc-characterization/ [29 April 2008]

27. X.891: Information technology - Generic applications of ASN.1: Fast infoset. http://www.itu.int/rec/T-REC-X.891-200505-I/en/ [5 May 2009]
28. Xerces-C++ page. http://xerces.apache.org/xerces-c/ [7 April 2009]
29. XML 1.0 Specification page, section 1.1. http://www.w3.org/TR/2006/REC-xml-20060816/ [29 April 2008]