Qualifying Exam Research Report, May 2007

A Framework for Understanding The Portability of C Types

Marius Nita
University of Washington
marius@cs.washington.edu

Abstract

The C standard does not specify the memory layouts of types, opening the door for portability problems that arise from discrepancies in how various platforms choose to represent types in memory. I report on the design and implementation of a framework that provides automated help for finding and understanding these discrepancies. A dynamic analysis gathers *memory layout constraints* on types. An offline analysis checks the truth of these constraints against two abstract *platform descriptions*. The *host* description models the platform on which the program has been developed, tested, and assumed to work as intended. The *target* description models the platform on which the program is to be (perhaps hypothetically) ported. A warning is issued when a type may be used in a way that violates its associated layout constraints on the target platform but *not* on the host.

I envision the framework being used in two key ways: to find type-portability bugs in C programs, by checking the program's constraints against platform descriptions that model concrete target platforms of interest, and to understand the type-portability of programs, by checking constraints against many possibly unrealistic descriptions and observing the platform features that cause the constraints to become false. This process helps programmers discover and understand portability boundaries: what problematic assumptions the code makes, the locations where these assumptions arise, and consequently the kinds of platforms on which the program behaves as intended.

1. Introduction

Previous efforts on assigning formal semantics to C and C-like languages have ignored the issue that C is, by definition [25], *platform-dependent* (where "platform" is taken to mean the C compiler plus its target architecture). The implementor of the compiler is free to choose how to represent features that are left *unspecified* by the language definition, and the programmer is capable of writing programs that make *assumptions* about platform idiosyncrasies and therefore work as intended only when compiled with particular compilers. Previous work has either tacitly assumed a particular platform, has completely avoided modeling platform-dependent behavior, or has classified implementation-dependent expressions as meaningless [32]. The chief drawback of these formulations is that they do not help us reason about C code *in practice*, where these assumptions are made on purpose. At a practical level, this means that there is no sound foundation from which to build analyses that separate programs making incorrect assumptions from ones that do not.

In previous work [31], we developed a theoretical model for C-like languages that takes into account the notion of platform dependency by parameterizing the semantics with an explicit platform. The work presented in this paper leverages those results in the design and implementation of a tool that allows reasoning about the portability of C type declarations across platforms. For example, the in-memory representation of data of type `struct T{char x;int y;double z;}` depends on the sizes and alignments of `int` and `double`, the alignment of structs, and the algorithm for recursively laying out structs. All these features are left unspecified by the C language definition and each C compiler may make different decisions, typically depending on the features of its underlying machine.

The notion of portability addressed by the present work is at the level of types and memory safety. It may classify a dependency as "bad" on a platform if it may lead to a memory safety violation. For example, a program may run as intended on one platform but fail with an alignment fault on another due to an incorrect assumption about the latter platform's type layout policy and memory fetching abilities. This notion of portability does not capture dependencies that go beyond memory safety. For example, if a program prints "0xaabbccdd" on one platform and "0xddccbbaa" on another due to a difference in endianness, but is memory-safe on both, our system will not necessarily issue a warning. Similarly, if the target platform is missing a needed header file or a system call, our system will not necessarily help. This is due first to our focus on unspecified "holes" in the C language, and second due to the difficulty of handling full equivalence-portability (in the sense that two programs evaluate the same on two platforms) because of hard problems like modeling overflow and floating-point roundoff behavior. We believe that our basic framework can be extended in both of these directions with enough effort. Work on equivalence-portability is currently underway.

2. Background

One of the key promises of high-level languages such as Java [8] and Scheme [13], is that they are *portable*: they aim to guarantee that programs behave the same on all architectures for which respective interpreters or compilers exist. This is achieved by disallowing language implementations from choosing how to represent particular features: the sizes of types, endianness, overflow behavior, floating-point roundoff behavior, evaluation order of function arguments, are all implemented such that they behave the same on all platforms. This flavor of "enforced" portability disallows programmers from discovering features of the underlying architecture and resolutely rules out the class of programs that use this functionality. As a consequence, many system-level programs, such as operating system kernels, device drivers, runtime systems, and even implementations of high-level portable languages, are written in low-level, unchecked, unportable languages like C. Portable languages themselves recognize the need for this functionality and provide foreign function interfaces (FFI): ways to write C code with which a high-level program can communicate.

C, on the other hand, is *semi-portable*, in the sense that portable programs can be written, but so can wholly unportable ones, or ones that work as intended on a known set of C implementations and their architectures, but not necessarily on others. The C language specification [25] delegates crucial decisions to the language implementor for reasons of efficiency and control. The implementor is free to choose the sizes and alignments of types, the order of evaluation for function arguments, and the order in which bytes are stored in memory, among others. Typically, these choices reflect features of the underlying architecture, though this mustn't be the case. To write a semi-portable program in C, a programmer must understand the C specification, which features of the language are left up to the implementor, and how the target implementations handle these unspecified features.

At the time of this writing, to the best knowledge of the author, transitioning from portable to platform-specific code is an all-or-nothing endeavor. If a program has been developed and tested in Java or Scheme, and the programmer comes to a point where full portability is too strong and wishes to make a decision based on a feature of the underlying architecture, he must resort to the FFI and write the code in C. Besides his knowledge, reading the code, ad-hoc testing, and some very weak tool support (discussed in Section 8), the programmer has no other means to increase his confidence that this code is as portable as intended and no less. Worse, there exists no support for developing code with portability in mind. Verifying that code is portable, to the extent possible, involves direct access to all the target software and architectures. Finally, there exists no sophisticated tool support for porting low-level software to new platforms. Once again, programmers must rely on knowledge of the spec, knowledge of the language implementation, and testing.

The scarcity of development tools for portability does not imply an absence of a *need* for such tools. The problem of writing portable code in C is well-known and well-documented. Querying Amazon.com for related books yields many relevant results, e.g. [24]. Typically, such books contain recipes for writing portable code and a distilled version of the unportable subset of the C specification. For example, readers will be instructed to not rely on how the compiler lays out structures in memory and to make sparing use of the `sizeof` operator. Evidence of concern (and widespread misunderstanding of the C specification) on the part of developers is abundant on mailing lists [2, 1, 3, 11, 6], web sites dedicated to documenting common problems [10, 9], and papers describing experience with portability problems [36]. One well-documented [16], high-impact set of portability issues occurred in the mid- to late-90s, when a large set of software packages were identified to run incorrectly on the ARM architecture, and subsequently ported. Most of these issues were due to the ARM processors' limited ability to access data on non-4-byte boundaries. The programs in question assumed that data could be accessed at smaller boundaries. The existence of such literature serves as evidence that little to no adequate automated solutions exist, and its ubiquity confirms that the problem is of widespread interest.

The work presented in this paper is a serious attempt at providing support for detecting portability problems in and understanding the portability of C programs. I build on our previous work [31] to describe a practically-motivated model for portability, and then describe the implementation of a tool that can detect a class of portability problems described below.

2.1 Key idea

I define a "platform" to mean an oracle that knows the compiler's policy for laying out types in memory. The platform includes the sizes and alignments of types, the offsets of struct fields, and the algorithm for recursively laying out nested structs. Consequently, the notion of "portability" that I consider is concerned with differences in the layouts of types on different platforms. These differences can lead to undesired effects, some of which are drastic: a pointer cast

```
1   struct S { void *buf; int len; };
2   struct D { void *buf; size_t len; };

3   struct S ss[100];
4   struct D *ds = (struct D*)ss;

5   //treat ds[N].len as the length of ds[N].buf
```

Figure 1. Example buggy code

may be fine on the Gcc/X86 platform, but can lead to an unaligned fetch and even a computer crash on Gcc/ARM.

This type-layout portability focus leads to a key idea: We can define a language for describing memory layouts, and then view platforms as translators from C types to memory layouts. Given a C type and two platforms, we can translate the type on each platform and look for problematic differences among the two resulting memory layouts. These differences may be sources of portability bugs.

2.2 Outline of the approach

The basic technique assumes a scenario where a program has been developed, tested, and debugged on one platform (called the *host*), and the developer is now interested in porting the program to a new platform (called the *target*). Given a C program, we can extract a set of constraints about the layout compatibility of types. Roughly, these constraints express the conditions under which the layouts of types on the target don't cause problems that don't already exist on the host. The constraints can then be checked in the context of the host and the target and a warning is issued for each failed constraint, along with a code location where the problem occurred. Section 5 discusses the actual implementation in detail.

Figure 1 shows a piece of code with a portability bug in it. On systems where `sizeof(int) == sizeof(size_t)`, this code works as intended. On 64-bit platforms on which `sizeof(int) == 4` and `sizeof(size_t) == 8` (because `size_t` is a type alias for `long`), this code does *not* work as intended. On such a 64-bit platform, the type `struct S` is laid out as an eight-byte block (for `buf`), followed by a four-byte block (for `len`), followed by four bytes of padding. The type `struct D` is laid out as two back-to-back eight-byte blocks. Treating the array `ss` as an array of `struct D` is always a bug on big-endian 64-bit systems and a bug on 64-bit little-endian systems if the pad bytes on the layout of `struct S` are not all-zero.

Our approach proceeds by generating a constraint on line 4 stating that the layout of `ss` must be compatible with the layout of an array of `struct D`. Given a 32-bit *platform description* for the host and a 64-bit one for the target, an analysis obtains the layouts of `struct S[]` and `struct D[]` on the host and then on the target. The layouts are then passed to a *layout subsumption* algorithm that determines whether the former layout can be accessed through a type with the latter layout. The algorithm returns false and issues a warning, pointing the programmer to the offending line 4. Platform descriptions are software implementations of their concrete counterparts. They are easy to implement because we only model a small subset of a platform, namely its policy for laying out types in memory.

The rest of the paper is organized as follows. The next two sections give an abstract overview of the system's key ingredients. Section 5 discusses the implementation. Section 6 describes two case studies of running the tool on real-world software. Section 7 expands on the implementation discussion and details limitations of the approach. Section 8 describes related work and Section 9 concludes.

$$\textsc{ptr}\quad \frac{\alpha_1 = \alpha_2 \times i}{P; \delta \vdash \mathrm{ptr}_{\alpha_1}(\overline{\sigma}_1 \overline{\sigma}_2) \leq \mathrm{ptr}_{\alpha_2}(\overline{\sigma}_1)} \qquad \textsc{unroll}\quad \frac{P.\mathtt{xtype}\ \delta\ (\delta\ \nu) = \overline{\sigma}}{P; \delta \vdash \mathrm{ptr}_{\alpha}(\nu) \leq \mathrm{ptr}_{\alpha}(\overline{\sigma})} \qquad \textsc{roll}\quad \frac{P.\mathtt{xtype}\ \delta\ (\delta\ \nu) = \overline{\sigma}}{P; \delta \vdash \mathrm{ptr}_{\alpha}(\overline{\sigma}) \leq \mathrm{ptr}_{\alpha}(\nu)} \qquad \textsc{pad}\quad \frac{\mathtt{size}\ P\ \sigma = i}{P; \delta \vdash \sigma \leq \mathrm{pad}[i]}$$

$$\textsc{add}\quad \frac{}{P; \delta \vdash \mathrm{pad}[i]\mathrm{pad}[j] \leq \mathrm{pad}[i+j]}$$

Figure 2. Layout subsumption ($P; \delta \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$), omitting standard rules

3. System Overview

This section elaborates the key ideas underlying the constraint system. I begin by presenting a language for describing platform-independent memory layouts, followed by platform descriptions. I then describe a notion of *physical subtyping* for memory layouts, which will later be instrumental in checking the portability of pointer casts. Finally, I discuss the constraint language and how constraints are checked.

3.1 A language for memory layouts

The concept of memory layout is pervasive in informal discussion about the C language. For example, the statements "the size of `long` on 64-bit SPARC is eight bytes" and "ARM pads structures up to the nearest four-byte boundary" make reference to a notion of memory layout including features like padding, alignment, and the amount of space data occupies. These features are made explicit in the following language:

$$\begin{array}{rcl} \sigma & ::= & \mathrm{byte} \mid \mathrm{pad}[i] \mid \mathrm{ptr}_{\alpha}(\overline{\sigma}) \mid \mathrm{ptr}_{\alpha}(\nu) \\ \nu & \in & \text{type names} \\ i, \alpha & \in & \mathbb{N} \end{array}$$

σ denotes an atomic layout unit and $\overline{\sigma}$ denotes a sequence of such units. byte represents a byte of data, $\mathrm{pad}[i]$ is i bytes of padding, and $\mathrm{ptr}_{\alpha}(\overline{\sigma})$ represents a pointer to an α-aligned memory layout $\overline{\sigma}$. $\mathrm{ptr}_{\alpha}(\nu)$ is the same, except it is a pointer to a named type with name ν. Pointers to named types are necessary to model C-style recursive types: a `struct` can refer to itself only through a pointer to its own name.

As an example, consider the type `struct{short x;}*`. On x86 using the Gcc compiler with no special command-line flags, its layout is $\mathrm{ptr}_2(\mathrm{byte\ byte})$. Using Gcc on an ARM platform, the layout is $\mathrm{ptr}_4(\mathrm{byte\ byte\ pad}[2])$. Due to the ARM's limited support for fetching data on non-4-byte boundaries, Gcc 4-byte aligns all structs and inserts trailing padding accordingly. To make precise this notion of type layout on a particular platform, we need a way to talk about platforms.

3.2 Platform descriptions

When looking to understand the memory layout of a type on a platform, we are typically only interested in a small subset of that platform's features, such as the sizes and alignments of base types and the algorithm for recursively laying out nested structs. This type-level abstracted representation is captured by the following signature:

$$\begin{array}{rcl} \mathtt{ptrsize} & : & \mathbb{N} \\ \mathtt{alignof} & : & \delta \to \tau \to \mathbb{N} \\ \mathtt{offsetof} & : & \delta \to f \to \mathbb{N} \\ \mathtt{xtype} & : & \delta \to \tau \to \overline{\sigma} \end{array}$$

$$\begin{array}{rcl} \delta & \in & \nu \to \tau \\ \tau & \in & \text{C types} \\ f & \in & \text{field names} \end{array}$$

The `ptrsize` component gives the size of pointers. For simplicity, we choose not to support multiple pointer sizes, as in systems with "near" and "far" pointers. In current practice, most platforms have a consistent pointer size.

The rest of the components are parameterized by a mapping from names to C types (δ), which is necessary to resolve names inside compound types. Given a set of declarations, `alignof` takes a C type to its alignment in memory. `offsetof` resolves offsets of struct fields, assuming uniqueness of field names.[1] The `xtype` function is the most interesting of the bunch, as it encodes the policy for laying out C types in memory. It is essentially a "type compiler" targeting the layout language $\overline{\sigma}$.

A *platform description* is then defined as an implementation of this signature: a record of four functions. If *x86* and *sparc* are descriptions modeling their concrete counterparts, we expect ($\mathit{x86}.\mathtt{xtype}\ \ \delta\ \ \tau$) to yield a different memory layout than ($\mathit{sparc}.\mathtt{xtype}\ \delta\ \tau$) in general.

Worthy of note is the omission of a `sizeof` component, which exists as a platform-dependent operator in C. `sizeof` is not necessary and can be recovered from the existing signature. To take the size of a C type on a platform P, we translate it to its memory layout $\overline{\sigma}$ on P and then calculate the size of $\overline{\sigma}$ with the aid the following function:

$$\mathtt{size}\ P\ \sigma \quad = \quad \begin{cases} 1 & \text{if } \sigma = \mathrm{byte} \\ i & \text{if } \sigma = \mathrm{pad}[i] \\ P.\mathtt{ptrsize} & \text{if } \sigma \in \{\mathrm{ptr}_{\alpha}(\nu), \mathrm{ptr}_{\alpha}(\overline{\sigma})\} \end{cases}$$

3.3 Layout subsumption

It is common practice in C programming to use the memory layout of a type as if it were the layout of another type. For example, a pointer to `struct T{int x;int y;}` can be cast to a pointer to `struct U{int x;}`. After the cast, all reads and writes through `U` affect exactly the "`int x;`" prefix of `T`'s layout, which is memory-safe. Using our layout language, we can define a subsumption relation that makes precise this notion of layout compatibility.

Figure 2 shows the layout subsumption relation as a set of inference rules. The relation $P; \delta \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$ says that given a platform P and a set of declaration δ, the layout $\overline{\sigma}_1$ can be treated as if it were layout $\overline{\sigma}_2$. The key rule is PTR, which determines when a pointer to a layout can be treated as a pointer to a different layout. The rule allows "dropping the suffix" under the pointer, thus capturing the programming paradigm discussed in the previous paragraph. Another important rule, PAD, states that any sequence can be treated as a sequence of padding of the same length. Since pad bytes can't be directly written to or read from, this is always safe. The ADD rule says that two adjoining padding sequences can be intuitively seen as one contiguous pad sequence. The UNROLL and ROLL rules are administrative, invoked whenever a pointer to a name is encountered. Standard rules for reflexivity and transitivity are omitted.

[1] Uniqueness is easily enforced, renaming if necessary.

Notice that the PTR rule requires the alignment on the left to be a multiple of (or possibly equal to) the alignment on the right. This in effect encodes subsumption on alignments, which permits a larger layout subsumption relation than if these alignments were required to be equal. For example, it is always safe to treat a pointer to a 4-byte aligned value as if it were a pointer to a 2-byte aligned or unaligned value, which a limited PTR rule would disallow.

3.4 Constraint language

With memory layouts and platform descriptions in hand, we can express and check memory layout constraints. A constraint is a logical formula whose truth can be evaluated only in the context of a platform. Thus, a constraint can be viewed as a filter that splits the space of all platforms into those that make it true and those that make it false. In the next section we will craft constraints that we can extract from C programs, such that if a platform makes a constraint false, we can relate it to a possible type-portability bug (a layout incompatibility leading to a safety violation) on that platform; and if it makes it true, we can be confident that there is no safety violation.

We formulate the constraint language as a standard first-order theory, with universal and existential quantification, where quantified variables cannot range over predicates or functions. We augment the base theory with a number of custom function symbols and corresponding sorts. The new sorts correspond to the syntactic classes introduced in previous sections: $\mathbb{N}, \tau, f, \overline{\sigma}$. The new function symbols are as follows:

$$\begin{array}{rcl}
\texttt{h\_ptrsize}, \texttt{t\_ptrsize} & : & \mathbb{N} \\
\texttt{h\_alignof}, \texttt{t\_alignof} & : & \tau \rightarrow \mathbb{N} \\
\texttt{h\_offsetof}, \texttt{t\_offsetof} & : & f \rightarrow \mathbb{N} \\
\texttt{h\_xtype}, \texttt{t\_xtype} & : & \tau \rightarrow \overline{\sigma} \\
\texttt{h\_sizeof}, \texttt{t\_sizeof} & : & \tau \rightarrow \mathbb{N} \\
\texttt{h\_subtype}, \texttt{t\_subtype} & : & \overline{\sigma} \rightarrow \overline{\sigma} \rightarrow \{\text{true}, \text{false}\}
\end{array}$$

Functions prefixed by h_ intuitively refer to the host platform and those prefixed by t_ refer to the target. The first eight symbols correspond directly to the features of a platform description. h_sizeof and t_sizeof calculate the size of a type on the host and target, respectively. h_subtype and t_subtype demand that the first argument is layout-compatible to the second.

The constraint language can be used to express rich relationships between the host's and the target's type layout policies. As a simple example, to demand that host pointers are equal in size to target pointers, we can simply write ($\texttt{h\_ptrsize} = \texttt{t\_ptrsize}$). If we want to ensure that the host and the target lay out types identically, we write $\forall\tau.\texttt{h\_xtype}(\tau) = \texttt{t\_xtype}(\tau)$. More elaborate examples will be discussed in the next section.

The truth of first-order formulas that are not tautologies can be evaluated only in the context of a *model*. For example, a formula $f(1) \wedge g(2)$ is meaningless unless we can appeal to an model that gives meaning to f and g. In our context, the model is a triple (P_h, P_t, δ), where P_h describes the the host platform, P_t the target, and δ is a set of C type declarations. Given such a model, we can determine the truth of any formula written in the first-order theory described above. Applications of our custom function symbols are resolved as follows (showing only the host symbols):

Function	Interpretation under (P_h, P_t, δ)
h_ptrsize	P_h.ptrsize
h_alignof τ	P_h.alignof δ τ
h_sizeof τ	$\sum_{i=1}^{n}$ size P_h σ_i (where P_h.xtype δ $\tau = \sigma_1 \ldots \sigma_n$)
h_offsetof f	P_h.offsetof δ f
h_xtype τ	P_h.xtype δ τ
h_subtype $\overline{\sigma}_1$ $\overline{\sigma}_2$	$P_h; \delta \vdash \overline{\sigma}_1 \leq \overline{\sigma}_2$

The target symbols are resolved in the same manner, using P_t instead of P_h. Typical first-order formulas are resolved in the usual way. For example, letting C range over formulas, $C_1 \wedge C_2$ is true when C_1 is true *and* C_2 is true. Formulas quantified by variables that range over types or their translations are resolved by appealing to δ. For example, a formula $\forall\tau.C$ is true when for all types τ' that are base types or are in the range of δ, C with τ substituted for τ' is true.

Assuming that the platforms are well-behaved and implement total functions, that layout subsumption is decidable, and that formulas over the natural numbers use only addition, the constraint language presented so far is *Presburger arithmetic* [21, 33], which is known to admit a decision procedure in general. Though I have not formally proved that subsumption is decidable, I have implemented an algorithm and am confident in its correctness.

4. Constraints from C Programs

I now leverage the toolset presented in the previous section to show a set of interesting type-portability constraints that can be extracted from C programs. I assume the existence of a run-time type (RTT) analysis that assigns to each expression the type that it may have at runtime. Such an analysis is needed because we are interested in discovering layout incompatibilities that may occur at run-time, and since C's type system allows arbitrary pointer casts, the static type of an expression is at best a weak indicator of what its type may be at run-time.

Figure 3 shows the list of constraints currently supported by the system, assuming a run-time type analysis RTT(-). Constraint (1) and (2) are identical, and extracted from occurrences of sizeof operators. The latter refers to expression forms sizeof(e), in which case the size of e's type is always statically known, so the RTT result wouldn't help. They evaluate to false if the type passed to sizeof (or the type of the expression passed to sizeof, in the latter case) has trailing padding on one platform but not on the other. It is a common mistake to assume that a struct will be aligned according the alignment of its maximally aligned field. Some platforms (e.g. Gcc on ARM) align structs to a boundary that can be greater than that of any of its fields, which can result in unanticipated trailing padding.

Constraint (3) is extracted from pointer-to-pointer casts and expresses that if the two types involved in the cast are compatible on the host platform, then they must be compatible on the target. The constraint makes the reasonable assumption that the programmer intended that the types involved in the cast be layout-compatible, and so that layout compatibility should be preserved on the target. Constraint (4) is similar to (3) but encodes *suffix-cast*. For example, if e has type struct{char x;int y;}*, the casts (int*)&e->y and (struct{int z;}*)&e->y are legal suffix-casts on many platforms. The constraint says that if a suffix cast is legal on the host, then it must be legal on the target. $\gcd(-,-)$ is the greatest common divisor function. If a piece of data is n-byte aligned, then data starting at an offset of m bytes from its beginning is $\gcd(n, m)$-byte aligned.

5. Implementation

I have implemented a tool that puts to work the ideas presented in the previous two sections. Given a preprocessed C program, a dynamic analysis (implemented as an instrumentation phase on top of CIL [29] and a runtime system) is used to gather constraints. An offline *constraint analysis* (roughly a way to evaluate first-order formulas mechanically, along with an implementation of the layout subsumption algorithm) inputs the constraints and outputs a list of warnings and code locations regarding failed constraints. As described in Section 3.4, the truth of a constraint can be determined

	Expression	RTT analysis	Generated constraint
(1)	$\texttt{sizeof}(\tau)$		$\exists\sigma_{h1},\ldots,\sigma_{hn},\sigma_{t1},\ldots,\sigma_{tm}.$
(2)	$\texttt{sizeof}(e:\tau)$		$(\texttt{h\_xtype}(\tau)=\sigma_{h1}\ldots\sigma_{hn}$
			$\wedge\ \texttt{t\_xtype}(\tau)=\sigma_{t1}\ldots\sigma_{tm}$
			$\wedge\ \exists i.\sigma_{hn}=\mathrm{pad}[i]\Rightarrow\exists j.\sigma_{tm}=\mathrm{pad}[j]$
			$\wedge\ \exists i.\sigma_{tm}=\mathrm{pad}[i]\Rightarrow\exists j.\sigma_{hn}=\mathrm{pad}[j])$
(3)	$(\tau*)e$	$\mathrm{RTT}(e)=\tau'*$	$\texttt{h\_subtype}(\texttt{h\_xtype}(\tau'*),\texttt{h\_xtype}(\tau*))$
			$\Rightarrow\texttt{t\_subtype}(\texttt{t\_xtype}(\tau'*),\texttt{t\_xtype}(\tau*))$
(4)	$(\tau*)\texttt{\&}e\ \texttt{->}f$	$\mathrm{RTT}(e)=\tau'*$	$\exists\overline{\sigma}_1,\overline{\sigma}_2,\alpha,\overline{\sigma}'_1,\overline{\sigma}'_2,\alpha'.$
			$(\texttt{h\_xtype}(\tau'*)=\mathrm{ptr}_\alpha(\overline{\sigma}_1\overline{\sigma}_2)$
			$\wedge\ \texttt{h\_offsetof}(f)=\texttt{h\_sizeof}(\overline{\sigma}_1)$
			$\wedge\ \texttt{h\_subtype}(\mathrm{ptr}_{\mathrm{gcd}(\alpha,\texttt{h\_offsetof}(f))}(\overline{\sigma}_2),\texttt{h\_xtype}(\tau*)))$
			$\Rightarrow$
			$(\texttt{t\_xtype}(\tau'*)=\mathrm{ptr}_{\alpha'}(\overline{\sigma}'_1\overline{\sigma}'_2)$
			$\wedge\ \texttt{t\_offsetof}(f)=\texttt{t\_sizeof}(\overline{\sigma}'_1)$
			$\wedge\ \texttt{t\_subtype}(\mathrm{ptr}_{\mathrm{gcd}(\alpha',\texttt{t\_offsetof}(f))}(\overline{\sigma}'_2),\texttt{t\_xtype}(\tau*)))$

Figure 3. Currently supported portability constraints. The first column shows the expression from which the constraint is extracted. The "RTT Analysis" column shows the result of an analysis RTT(-) that determines the run-time type of an expression. A conservative RTT(-) may return many possible types, in which case we generate a constraint for each type.

only in the context of a model that includes a host and target platform and a set of type declarations. In the implementation, the set of type declarations are extracted from the input program and the host and target platforms are drawn from a *platform "database"* that contains many platform descriptions. The descriptions have been written by me as records of OCaml functions implementing roughly the signature described in Section 3.2. Adding descriptions is easy, as new descriptions can be built by extending old ones.

The implementation is unsound and incomplete: it can yield both false positives and false negatives (per run). Unsoundness is in part due to an assumption that all the fields of all structs are read/written, which mustn't be the case in general. E.g., if the code in Figure 1 never uses the `len` field to bound access to the corresponding buffer, there is no bug. Per-run incompleteness is because the tool can be used on as little as one file. A set of instrumented files can be linked with other plain C files, so all the runtime information gathered from the program refers to only the instrumented subset. If bugs exist in uninstrumented parts of a runtime trace, they won't be discovered. I view the ability to perform partial instrumentation as a feature.

In the rest of this section I give an overview of the tool's main components: the instrumentation phase, the runtime system, the constraint analysis, and the platform description database.

5.1 Instrumentation

The instrumenter takes a preprocessed C program and outputs a version of the program that records runtime constraint information on the side. To represent types at run-time, it uses a 1-1 mapping between types and integers, calculated during instrumentation. Since many files in a program may share the same types, and thus must use the same type numbers, the mapping is dumped to disk after a file is instrumented, and is loaded and updated when another file (in the same program) is instrumented. The instrumentation consists of inserting into the program calls into a custom C library, against which the program will be eventually linked. The library maintains an internal data structure mapping addresses to their types. The interface includes calls for registering the type of an address (`__reg_ptr`, `__reg_addrof`, `__reg_array`), for forgetting the type of an address (`__unreg`), for registering a pointer cast (`__reg_cast`), and for registering the run-time occurrence of a `sizeof` instruction (`__reg_sizeof`).

Supposing $\mathcal{N}(-)$ is the unique mapping assigning integers to types, Figure 4 gives an overview of how code is instrumented:

1. We instrument all dynamic allocation sites to insert the allocated pointer into the type mapping. We assume `malloc`-style allocators where the argument contains the size of the memory chunk to be allocated, in bytes. In addition to the address and the type, we pass into the runtime the size of the type (`__tmp_0`) and the number of bytes requested (`__tmp_1`). This allows the runtime to determine when an array is being allocated, by checking if the requested size is a multiple of the type's.
2. We instrument local and global declarations for variables whose addresses are taken anywhere in the program, to handle pointer casts such as `(T*)&x`. For local variables, we instruct the runtime to remove the mapping for `&x` when control reaches the end of the block.
3. We instrument C array declarations, since arrays in C behave like pointers and can participate in pointer casts. We unregister local arrays at the end of the block, as above.
4. We instrument all pointer cast sites by adding a call to pass into the runtime the source pointer and the destination type. The runtime can then determine the actual cast by looking up the type associated with the source pointer in its internal mapping. Note that after the cast, `x` contains the address to which `e` evaluates, so `x` contains the source address, though it may misleadingly look like the target.
5. We instrument all occurrences of `sizeof(T)`, requesting that the runtime records the type `T`. Note that in `sizeof(e)`, the size of `e`'s type is statically known. No run-time type lookup is performed.
6. We instrument all deallocation sites by inserting a call instructing the runtime to remove the deallocated pointer from the type map.

Once every file in the program is instrumented, all the files are linked against the custom runtime system, which implements all the custom calls in the instrumented code.

```
       Original code                Instrumented code
(1)    x = (T*)malloc(e);           __tmp_0 = sizeof(T);
                                    __tmp_1 = e;
                                    x = (T*)malloc(__tmp_1);
                                    __reg_ptr(x, 𝒩(T*), __tmp_0, __tmp_1);

(2)    {... T x; ...}             { ... T x;
                                    __reg_addrof(&x, 𝒩(T*));
                                    ...
                                    __unreg(&x); }

(3)    {... T x[C]; ...}          { ... T x[C];
                                    __reg_array(x, 𝒩(T[]), C);
                                    ...
                                    __unreg(x); }

(4)    x = (T*)e;                   x = (T*)e;
                                    __reg_cast(x, 𝒩(T*));

(5)    ... sizeof(e) ...;           ... sizeof(e) ...;
                                    // T is e's statically known type
                                    __reg_sizeof(T);

       ... sizeof(T) ...;           ... sizeof(T) ...;
                                    __reg_sizeof(T);

(6)    free(x);                     free(x);
                                    __unreg(x);
```

Figure 4. Instrumentation overview. $\mathcal{N}(-)$ is a function assigning a unique number to each type.

5.2 Runtime system

The runtime system is a C library whose chief responsibility is to keep track of runtime *constraint facts* and make them available to the later analysis stage. A constraint fact is a type number for `sizeof` constraints and pairs of type numbers for pointer cast constraints.

To keep track of pointer cast facts, the runtime maintains a precise mapping from pointers to their run-time types. The run-time type of a pointer is decided once and for all at allocation time, when the instrumented code invokes `__reg_ptr`, `__reg_array`, or `__reg_addrof`. Though at run-time the program will be able to view a pointer at a different type, the true type of the pointer (and hence that true *memory layout* it points to) does not change through execution, except in one case: the pointer is `free`d and then returned by a subsequent call to `malloc` at a different type. I.e., `malloc` may reuse an address once it's back on its free list.

The runtime has powerful support for non-atomic types such as arrays and (arbitrarily nested) structs. If we look up an address that points to an element in the middle of an array, which was never explicitly entered into the mapping, the look-up will still succeed and return the array element type. Likewise for pointers inside structs. This is achieved by using an interval tree. Upon allocating an array or a struct, the runtime records a mapping from the corresponding pointer to an interval whose lower bound is the pointer itself and whose upper bound is the pointer plus the size of the array/struct. When an interval lookup succeeds, the runtime resolves the type of the pointer by looking up the type of the base and performing the necessary offset calculations.

Eventually, the runtime system needs to dump constraint facts to disk for subsequent analysis. Because run-time facts may be registered very frequently, dumping them to disk as they occur may use an unacceptable amount of disk space and slow the instrumented program significantly due to I/O overhead. The runtime buffers a preset number of constraints (currently set to 1000) before dumping them. Upon constraint registration, if a constraint exists in the buffer already, it is ignored. A simple hashing scheme is used to check for duplicates in amortized constant time. Currently, the runtime dumps the whole buffer to disk when it fills up. Another possibility is to dump only a small (possibly randomly selected) subset of the buffer, to reduce possibly large stall times.

Finally, not shown in Figure 4, runtime calls take as additional arguments the location of the line of the code being instrumented, in the form of a file name and a line number. This information is recorded in the output file for every pointer cast and `sizeof` fact and used by the constraint analysis to link failed constraints to the offending code.

5.3 Constraint analysis

Given a list of run-time cast and `sizeof` facts produced by running an instrumented program, and descriptions for the host and target platforms, an offline analysis generates the corresponding constraints and checks their validity. The analysis is implemented in OCaml and uses CIL to process C types. Type numbers are resolved back into types using the offline map generated by the instrumenter.

Given a `sizeof` occurrence, the analysis generates a corresponding constraint as shown in entries (1) and (2) of Figure 3. Given a pointer cast, it generates a constraint as shown in entries (3) and (4) of the same figure. These constraints are then checked against the host and target descriptions and the list of type declarations generated during instrumentation.

The `t_subtype` and `h_subtype` constraints appeal to the subsumption relation, as discussed in Section 3.3. Part of the analysis is an algorithm deciding inclusion in the relation: given two layouts, a platform and a set of declarations, it returns true if the latter subsumes the former and false otherwise.

The layout compatibility relation defines "upcast" (forgetting suffix under pointer, subsuming to pad bytes, etc.) but does not handle pointer casts in the other direction, for example a cast from `char*` to a pointer to `struct T{char x;int y;}`. This type of downcast is typical in some programs, networking software in particular. For example, a `char` buffer is read off the wire but the programmer knows it represents a `struct T`, so he performs the cast and subsequently treats the buffer as a pointer to `struct T`. Downcasts are typically ignored by the analysis, since their corresponding constraints are generally false on both the host and the target. In some cases, however, a useful warning can be issued. Before ruling out a downcast as illegal, the constraint checker first verifies that if the cast's *alignments* are compatible on the host, they are also compatible on the target, and warns if they are not. This special case helps in finding alignment-related problems in software that performs the type of cast described above. Though the `char` buffer may indeed contain data that is layout-compatible to a `struct T`, if the buffer is unaligned and the struct is, an alignment error could occur when accessing the data through `struct T`.

The rest of the analysis appeals to the individual platform descriptions to resolve symbols and check constraints.

5.4 Platform descriptions

Platform descriptions are records of OCaml functions, roughly implementing the signature described in Section 3.2. The central component of a platform description is the implementation of the `xtype` function, which takes a C type to its memory layout on a particular platform. A typical `xtype` implementation (e.g. one describing Gcc on x86 without compilation flags) appeals to the `alignof` and `sizeof` components to lay out types and insert padding accordingly. Struct types are laid out by recursively laying out the components and concatenating the results.

The current set of platform descriptions have been implemented by me by hand. To increase my confidence that the code is correct, I've devised a way to test platform descriptions against their concrete counterparts. Given a set of type declarations, a tool generates a program that, for each type, prints its size and in the case of structs, the name and offset of each field. This information is loaded and compared to what `xtype` and `offsetof` (which is implemented in terms of `xtype`) return for the same types and fields. Using this approach, I found and fixed an important bug in the Gcc-x86 platform description.

New platforms can be easily added by subclassing current platforms. Since most platforms respect the layouts of atomic types when laying out structs, adding a new platform typically means subclassing an existing base platform and overriding some information regarding atomic types, without changing the basic layout algorithm. Currently, all platforms subclass a base platform (which consists of 150 lines of OCaml counting whitespace and comments) and are typically less than 10 lines of code.

6. Case Studies

As a preliminary evaluation of the tool, I've applied it to two C programs. Spread [14] is a high-performance fault-resilient messaging service, intended to be used as a general-purpose messaging bus for distributed systems. The Python interpreter [12] is the reference implementation of Python, a widely-used general purpose scripting language. I chose these two applications because Spread contains a real portability bug and Python contains a good example of semi-portable code: it breaks on platforms that make slightly unusual decisions when laying out types. I was aware of both of these issues before the tool discovered them.

6.1 Experience with Spread

I instrumented the full Spread program set (a library and five executables) and exercised it in a number of simple ways. Essentially, I started the server, connected a few clients to it, and repeatedly sent messages between the clients. The runtime recorded 47 unique constraint facts. Of these, two were reported as possible problems. One was a real portability bug and the other was a false positive: a failed `sizeof` constraint in a memory initialization routine, where trailing pad bytes aren't an issue.

The real bug (a version of the bug shown in Figure 1) is due to a hidden assumption that `sizeof(int)` is equal to `sizeof(size_t)`, which isn't necessarily the case on some platforms. Spread defines the following type:

```
struct scat_element { char *buf; int len; };
```

It then creates an array of `scat_element` and performs a cast from this array to pointer to `struct iovec`, subsequently manipulating the array as if it were a `struct iovec`. The latter is a C library type, typically defined as follows:

```
struct iovec { char *buf; size_t len; };
```

On some 64-bit platforms, so-called "LP-64" platforms, integers are kept as four bytes long, but `long`s and pointers are eight bytes long. The type `size_t` is normally a type alias for `long`.

When the analysis is performed with a 32-bit host and an LP-64 target, the pointer cast is deemed safe on the host but fails on the target, violating constraint (3) in Figure 3. Letting byte^n mean a sequence of n bytes, the host layouts of both types are equal to:

$$\mathrm{ptr}_4(\mathrm{ptr}_1(\mathrm{byte})\mathrm{byte}^4)$$

Both `int` and `size_t` were translated to byte^4, since they are both four bytes long and equally aligned. Clearly, a layout subsumes itself, so the cast is safe. On the target, however, the translations are as follows:

`scat_element`	`iovec`
$\mathrm{ptr}_8(\mathrm{ptr}_1(\mathrm{byte})\mathrm{byte}^4\mathrm{pad}[4])$	$\mathrm{ptr}_8(\mathrm{ptr}_1(\mathrm{byte})\mathrm{byte}^8)$

The layout on the left is *not* subsumed by the one on the right: the subsumption relation does not allow pad bytes to be treated as data.

Viewing `scat_element`'s layout through `iovec` is clearly not a bug on 32-bit platforms. It *can* be a real bug on little-endian LP-64 platforms, and it is *always* a real bug on big-endian LP-64 platforms. On little-endian LP-64 platforms, it is not a bug if the trailing pad bytes in `scat_element`'s translation are all set to 0. Since the low-order bits of the two `len` fields line up correctly, viewing the layout $\mathrm{byte}^4\mathrm{pad}[4]$ through either `int` or `size_t` yields the same number. Platforms are free to store garbage data in pad bytes, however. On big-endian LP-64 platforms, the low-order bits of the two `len` fields no longer line up, so whether pad bytes are set to 0 is irrelevant. After the cast, the low-order bits of `scat_element`'s `len` field will be in the high-order half of `iovec`'s `len` field, so viewing the same layout through `int` and `size_t` will yield different values. Viewing it through `size_t` yields a very large number, leading to out-of-bounds buffer accesses. An example of a big-endian LP-64 platform is Gcc on the 64-bit SPARC architecture.

This bug was reported on the Spread mailing list [11] but it does not appear to have been fixed in subsequent versions, possibly due to low demand from LP-64 users.

6.2 Experience with Python

I instrumented and analyzed Python's `struct` code: a dynamically-loaded module that implements marshalling of Python objects to and from strings. Since the type portability analysis tool can operate on as little as one file, I instrumented only the relevant C code. A minor hand-modification to Python's `main` function was needed, to insert calls to initialize and destroy the tool runtime. During my experiments, the runtime recorded 24 unique constraint facts. When run on these facts, the constraint analysis reports one warning and no false positives.

The warning is a `sizeof` constraint violation, issued in the context of a usual 32-bit host platform and an unusual target platform that models a 32-bit ARM but decides to leave the `short` type unaligned. The Python marshalling code uses a number of macros to decide the alignment of primitive types. The macro intended to compute the alignment of `short` is:

```
struct st_short { char x; short s; };
#define SHORT_ALIGN (sizeof(struct st_short) \
                     - sizeof(short))
```

The warning was issued for `sizeof(struct st_short)`. On the host platform, the layout of `st_short` is $\mathrm{byte}\,\mathrm{pad}[1]\,\mathrm{byte}^2$, while on the target it is $\mathrm{byte}^3\mathrm{pad}[1]$. Only one of the layouts has trailing pad bytes, so constraint (1) in Figure 3 is violated.

Upon closer inspection, we notice that `SHORT_ALIGN` evaluates to 2 on the hypothetical platform, despite the fact that `short` is unaligned. The actual answer should be 1. ARM-like platforms align all structures on boundaries no smaller than four bytes and add trailing padding accordingly, something the macro doesn't account for. While the warning is meaningless for real target platforms, it is certainly a bug on the hypothetical one, and sheds light on the type portability assumptions made by the code. It is feasible (and within the limits of the C standard) for a platform to unalign `short`s while keeping structs 4-byte aligned.

7. Discussion and Limitations

This section discusses various drawbacks of the general approach described in this paper, with particular focus on the implementation, along with smaller issues not mentioned in the main discussion.

7.1 Dynamic analysis

In retrospect, a conservative static analysis would have worked better. The chief benefit of the dynamic analysis is the run-time type precision. When a pointer cast is recorded, we know *exactly* what the source type is, regardless of casts to `void*` and back or possibly numerous pointer-cast hops through other types. The drawbacks are many:

- A rich set of program inputs is assumed, so that the dynamic analysis can reach many program paths. Many real-world (especially open-source) programs do not enjoy such test harnesses. The user of the system is faced with learning how to force the program down as many execution paths as possible.
- Performing an evaluation of a large set of real-world programs is difficult. As mentioned above, many of these programs lack rich test harnesses. As a user who doesn't understand the innards of these programs, it is extremely difficult to know whether the generated inputs are exercising the right program paths. Path-coverage tools could be used to learn some of this information.
- Finding type-portability issues in a program is a multi-step process. The user must first instrument the program, exercise it many times in a rich test harness with no knowledge of whether he exercised a bug, and then invoke the analysis separately. It is possible to run the analysis in a separate process and have the runtime system communicate information to it directly via IPC, allowing errors to be reported in real-time.
- The dynamic analysis in its current form cannot be run on low-level code such as kernels and device drivers. To analyze such code, one would have to implement a custom runtime system in the kernel that outputs run-time information to a special user-visible filesystem, such as `/proc` on Linux, and ensure that the instrumentation doesn't violate the fine-grained assumptions that kernels make about how code is compiled. Another possibility is to run the kernel in userspace, or run device drivers in a thin userspace harness. Either way, the setup overhead in analyzing kernel code is large.
- Portability bugs, like many types of bugs, are likely to lie dormant on program paths that are not exercised very often by test harnesses or real-usage inputs. Thus, even in the presence of a rich test harness, a dynamic analysis would miss these bugs.

A static analysis would largely fix these problems at the expense of precision loss in run-time type information. Presumably a static analysis that approximates run-time types could assign many types to an expression, many of which would result in false positives. My experience with the tool tells me that the number of false positives would be manageable and fairly precise static analysis is possible. Such an analysis would also be of independent interest.

7.2 Alignment information

In its current form, the runtime system communicates type information to the constraint analysis, but does not communicate any runtime information, such as the addresses involved in a cast. This extra information could be used to eliminate false positives. For example, a pointer cast may fail because the alignment of the source type is lower than that of the destination type (e.g. casting `struct T{char x[4];}*` to `struct U{int x;}*`). However, if it indeed happens that the source type is properly aligned, a warning is a false positive. This kind of false positive could be ruled out if the analysis knew the source address. The tool can be easily augmented in this direction. It hasn't yet been done because false positives are not currently a problem.

7.3 Preprocessor

As mentioned in Section 5, the instrumentation and analysis occurs after the code has been preprocessed. Thus, any code that was eliminated via selective compilation is not considered in the analysis. This may seem suspicious at first, since the preprocessor is a standard way to make platform-dependent decisions and enforce portability.

The purpose of the tool is to discover portability problems on the host. Presumably, code that has not made it into the preprocessed program has already been assigned as belonging to a different platform. The tool aims to help the programmer find portability problems in code running on the host, so that he can properly tuck it away under platform-dependent preprocessor directives if needed. Should a programmer decide that a warning is unwarranted, because he has already fixed that particular problem by writing equivalent target-specific code guarded by a proper preprocessor directive, he could flag the warning as "fixed" so that it is not shown again. Such a mechanism for flagging warnings is feasible but not currently implemented.

7.4 Malloc wrappers

Since custom allocators and `malloc` wrappers are pervasive, our tool offers a command-line option to specify all the possible names for allocators and deallocators. We do make the assumption that allocators are `malloc`-style, taking as argument a number of bytes to be allocated. The tool treats allocators as primitive functions and does not instrument their bodies. We do not instrument inside `malloc` wrappers because they typically cast the `malloc` return to `void*` or `char*`, which means that our address-type mapping would contain *only* this type for all addresses!

7.5 Other kinds of portability problems

The tool's focus is on the portability of types. One could presume that issues like endianness (whether the least significant byte of a piece of data is at the highest or lowest of the memory addresses of the bytes in that data) should be included in type portability, since endianness is defined in terms of the layout of data. Our approach does not handle endianness-related issues because our focus resolutely ignores the contents of data. To capture endianness, we would need a way to separate one byte of layout from another, in other words reason about the contents of bytes.

While the system can be extended to detect some endianness-related portability problems as a special case, I view endianness as part of a more expressive framework that reasons both about the preprocessor and the contents of data. Portability in such a framework is defined as "the program yields the same observable final state on both the host and the target." Such a framework is left for future work.

8. Related Work

Previous work related to the ideas presented in this paper can be split into three categories: making C safer, employing a notion of layout subsumption to check the legality of pointer casts; other approaches to detecting portability problems in code; and instrumentation for detecting problems in C code.

8.1 Physical subtyping and safe C

CCured [28, 18, 27] is a C compiler that, among other techniques, employs a "physical subtyping" relation to determine when pointer casts are safe on the assumed platform. Their notion of physical subtyping is much like the layout subsumption relation in Section 3.3. CCured largely borrowed their notion of physical subtyping from work by Chandra *et al.* [17] and Siff *et al.* [34], who used it to build a tool for understanding pointer-cast patterns in C code. Our own notion of layout subsumption was heavily inspired by the Chandra/Siff work. However, their relation operates at a higher level and is less expressive: subsumption forces fields to line up exactly and their names to be equal. So while a cast from `struct T{int x;int y;}*` to `struct U{int x;}*` is allowed, a cast from `struct T{int x;}*` to `struct U{char a;char b;char c;char d;}*` is rejected. First, because the field names don't match, and second because the offset of `x` is not equal to the offset of `a`. These kinds of casts are common in C and our relation allows them.

CCured, along with Cyclone [22, 23, 26] and SafeCode [19], make C safe but don't address portability. After instrumenting the code for safety, they pass it down to a standard C compiler. If the input code performs a cast deemed illegal on the assumed platform, a runtime exception may be thrown. CCured is built on top of CIL [29], a C frontend that makes the same assumptions as the standard C compiler on that platform. Typically, at the time these compilers are built, their platform assumptions are fixed to those of the underlying C compiler.

8.2 Work on portability

Possibly most closely related to our work, the *GUARD* [35] debugger can be used to find portability problems in programs. Presumably, the user can have a program running on two platforms and *GUARD* can debug them in parallel assuming that the two platforms are networked. The programmer must specify points at which the two programs should be in equivalent states, and *GUARD* can detect when these specifications are violated. Using a form of *delta debugging* [37], *GUARD* can narrow in on the point when the programs became out of sync. Like our system, *GUARD* aims to help the programmer understand and find portability problems using a dynamic analysis. Unlike our system, it requires the presence of both the host and target platforms. Thus, it makes it very hard to perform portability checking against a large number of platforms, and impossible to check against hypothetical platforms. Another difference is that *GUARD* requires the programmer to provide annotations. Our system works on plain C.

The key service provided by *GUARD* is assistance in identifying the *location* where the states of the two programs became out of sync, letting the programmer understand and trace the true source of the problem. This means simultaneously analyzing both versions of the program. Our approach presents the programmer with a list of possible *sources* of problems along with explanations, letting him find the locations of actual problems, if any. Given this and that the programmer need not understand both versions of the program, the burden on the programmer is lighter in our system than in *GUARD*.

The Gnu C Compiler (Gcc) [5] includes flags that can be used for finding type portability problems. The `-Wpadded` flag warns whenever padding is included in a type's layout. The programmer can presumably use this information to increase his understanding of the underlying type layout policy. The `-Wcast-align` flag warns about pointer casts that may result in unaligned accesses. E.g., it warns about a cast from `char*` to `struct T{char x;}*` on the ARM platform, where the former type is unaligned and the latter is 4-byte aligned. Cross-compilation can be used to get help from Gcc without physical access to the target platform.

`-Wpadded` is a bare-bones approach to finding portability problems. Padding is a natural occurrence in layouts and most of the time is not indicative of problems. Our system doesn't care if padding is used, so long as it doesn't lead to type compatibility problems. `-Wcast-align` is useful in finding bugs on platforms with unconventional alignment policies, such as the ARM, but offers no help with other problems, like porting from 32- to 64-bit. Moreover, it warns only about types known by the Gcc type system, which is only a weak indicator of the actual types involved in the cast at runtime. Our approach faces none of these shortcomings, in additional to being much more general and extensible.

Microsoft Visual Studio [15] includes a flag that aims to find 64-bit porting bugs in C code. The check consists of scanning the program syntax and warning about all expression forms `(int)e` where `e` has pointer type. Since on the 64-bit platform pointers are 64 bits wide while `int` remains 32-bit, this sort of pattern is almost always wrong. Our notion of layout subsumption by definition doesn't handle conversion casts like the one above, since they concern the *contents* of data, not its layout. Our approach, however, handles many kinds of type portability problems that this simple check does not. The tool can be easily extended to special-case this type of bug.

Lint-like technology, such as Splint [20], uses static analysis to flag suspicious patterns in C code, including a range of unportable constructs. The analysis can be guided by programmer-supplied annotations in the form of special C comments. Portability bug-checking in these tools is ad-hoc and doesn't include a notion of platform.

8.3 C instrumentation for bug-finding

Tools like Valgrind [30] and Purify [7] work by instrumenting C programs to detect memory corruption errors. The user runs the instrumented C program on a number of inputs. If a memory error occurs during these runs, the tool offers detailed information about the cause and location. While their general architecture is similar to ours, Valgrind and Purify do nothing about portability problems.

Polishchuk *et al.* augmented the Gnu debugger to perform heap type inference irrespective of the programmer-declared types. Like us, they gather and solve layout constraints and use a notion of layout subsumption to determine when a layout can be viewed at a different type. (Their layout subsumption is borrowed directly from the Chandra/Siff work [17, 34] mentioned above.) Their focus is on finding memory corruption bugs; they do not address portability.

9. Conclusions and Future Work

We have presented a basic approach for understanding and checking the portability of types in a C program. We've described an implementation of the technique, which uses a dynamic analysis to extract relevant information from a running C program, and an offline analysis to check a set of portability constraints. We've described our experiences with two case studies, on the Spread [14] distributed system architecture and the Python [12] interpreter. Of the 71 constraint facts, the analysis reported three warnings, two of which were relevant issues and one a false positive.

At the top of the future work list is a static analysis that approximates run-time type information. To be useful in general, such an analysis would have to make use of data and control flow information to increase precision. E.g., a typical pattern in C code is to perform a cast to `void*` and at a later point perform a downcast from `void*`, knowing that the pointer in question points to a particular kind of layout. An analysis that uses flow information may determine that a downcast from `void*` is really an upcast.

With a static analysis in hand, the technique can be applied in a number of ways. One example is inclusion in an integrated development environment like Eclipse [4] or Microsoft Visual Studio. Through a graphical interface, the programmer may be able to select platforms to check the program against. A live analysis would highlight sources of problems and could easily display layout discrepancies between types, visually explaining why a cast works on the host but not on the target.

Another idea for future work is to *enforce* type portability by explicitly laying out types in a particular way. For example, given a pointer cast that works on the host and breaks on the target, the problem is to find "least common denominator" layouts for the two types such that the cast is forced to succeed on both platforms. This approach can be integrated into CCured-like systems to emit C code in which the types may be explicitly padded to force the underlying C compiler to lay them out a particular way. A program that targets CCured can then make more inadvertent assumptions, as the augmented CCured will fix them at compile time.

Acknowledgments Sam Guarnieri implemented an early version of the runtime system. Many ideas presented here were developed under the guidance of, and in collaboration with, Dan Grossman. Dan Grossman, Craig Chambers and David Notkin provided invaluable advice throughout the project.

References

[1] ARM: Alignment Bug. http://lists.linux-wlan.com/pipermail/linux-wlan-devel/2003-July/002557.html.

[2] ARM: Invalid Stack Alignment Bug. http://gcc.gnu.org/ml/gcc-patches/2000-03/msg00384.html.

[3] Bugzilla Bug 842. http://bugzilla.mindrot.org/show_bug.cgi?id=842.

[4] ECLIPSE: An Open Development Platform. http://www.eclipse.org.

[5] GCC: The GNU Compiler Collection. http://gcc.gnu.org.

[6] IBM HOTSPOT Portability Bug. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4160327.

[7] IBM Rational Purify. http://www.ibm.com/software/awdtools/purify.

[8] Java Technology. http://java.sun.com.

[9] Mozilla.org C++ Portability Guide. http://www.mozilla.org/hacking/portable-cpp.html.

[10] Processor Differences. http://www.windowsitlibrary.com/Content/1685/05/1.html.

[11] Spread: Portability bug on Solaris 8. http://commedia.cnds.jhu.edu/pipermail/spread-users/2002-November/001185.html.

[12] The Python Programming Language. http://www.python.org.

[13] The Scheme Programming Language. http://www-swiss.ai.mit.edu/projects/scheme.

[14] The Spread Toolkit. http://www.spread.org.

[15] Visual Studio Developer Center. http://msdn.microsoft.com/vstudio.

[16] *The ARMLinux Book Online,* Chapter 10. May 2005. http://www.aleph1.co.uk/armlinux/book.

[17] S. Chandra and T. Reps. Physical type checking for C. In *ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 66–75, Toulouse, France, Sept. 1999.

[18] J. Condit, M. Harren, S. McPeak, G. Necula, and W. Weimer. CCured in the real world. In *ACM Conference on Programming Language Design and Implementation*, pages 232–244, June 2003.

[19] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM Conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006.

[20] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. Lclint: a tool for using specifications to check code. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 87–96, New York, NY, USA, 1994. ACM Press.

[21] Fischer and Rabin. Super-Exponential Complexity of Presburger Arithmetic. In *SIAMAMS: Complexity of Computation: Proceedings of a Symposium in Applied Mathematics of the American Mathematical Society and the Society for Industrial and Applied Mathematics*, 1974.

[22] D. Grossman. Quantified types in imperative languages. *ACM Transactions on Programming Languages and Systems*, 28(3):429–475, May 2006.

[23] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *ACM Conference on Programming Language Design and Implementation*, pages 282–293, Berlin, Germany, June 2002.

[24] B. Hook. *Write Portable Code*. No Starch Press, 2005.

[25] *ISO/IEC 9899:1999, International Standard—Programming Languages—C*. International Standards Organization, 1999.

[26] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002.

[27] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *29th ACM Symposium on Principles of Programming Languages*, pages 128–139, Portland, OR, Jan. 2002.

[28] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, May 2005.

[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 *Computational Complexity*, pages 213–228, 2002.

[30] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 232–244, June 2007.

[31] M. Nita, D. Grossman, and C. Chambers. A Theory of Implementation-Dependent Low-Level Software (Technical Companion). Technical report, Univ. of Wash. Dept. of Computer Science & Engineering, July 2006. Available at http://www.cs.washington.edu/homes/marius/papers/tid/.

[32] M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.

[33] W. Pugh and D. Wonnacott. Eliminating false data dependences using the omega test. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 140–151, New York, NY, USA, 1992. ACM Press.

[34] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *7th European Software Engineering Conference and 7th ACM Symposium on the Foundations of Software Engineering*, pages 180–198, Toulouse, France, Sept. 1999.

[35] R. Sosic and D. Abramson. Guard: A relative debugger. *Software - Practice and Experience*, 27(2):185–206, 1997.

[36] H. Spencer and G. Collyer. #ifdef considered harmful or portability experience with C news. pages 185–198, Summer 1992.

[37] A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC / SIGSOFT FSE*, pages 253–267, 1999.