Preprint. Accepted for OOPSLA 2016.

Dynamically Diagnosing Type Errors in Unsafe Code

Stephen Kell

Computer Laboratory, University of Cambridge
Cambridge, United Kingdom
firstname.lastname@cl.cam.ac.uk

Abstract

Existing approaches for detecting type errors in unsafe languages are limited. Static analysis methods are imprecise, and often require source-level changes, while most dynamic methods check only memory properties (bounds, liveness, etc.), owing to a lack of run-time type information. This paper describes libcrunch, a system for binary-compatible *run-time* type checking of unmodified unsafe code, currently focusing on C. Practical experience shows that our prototype implementation is easily applicable to many real codebases without source-level modification, correctly flags programmer errors with a very low rate of false positives, offers a very low run-time overhead, and covers classes of error caught by no previously existing tool.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Debugging aids

Keywords C, unsafe code, pointers, type information

1. Introduction

C, C++ and other unsafe languages remain widely used. Here "unsafe" means that the language does not enforce memory- or type-correctness invariants. A general type-correctness invariant might be that *any access to a stored value reads or writes a value consistent with the type the storage is intended to hold*. In this paper we describe a system for dynamically checking this property. By contrast, existing tools generally focus on memory invariants, such as *all pointer uses respect the bounds of the object from whose address the pointer derives* (a memory property; spatial), or *an object is only reclaimed once no pointers into it may any longer be used by the program* (a memory property; temporal).

Many approaches exist for making unsafe languages less unsafe. Specifically regarding *type-correctness*, some dynamic analysis tools exist [3, 16] but suffer high run-time overhead. Hybrid static/dynamic systems [12, 21] sacrifice compatibility with existing code at either source level (entailing porting effort) or binary level (precluding use of prebuilt libraries). The most practically compelling tools such as Memcheck [24], SoftBound and CETS [18, 19] check only memory properties, and (by design) have no awareness of data types.

```
if (obj->type == OBJ_COMMIT) {                                  1
  if (process_commit(walker, (struct commit *)obj))             2
    return -1;                                                  3
  return 0;                                                     4
}                                                               5
```

Figure 1. Pointer cast in C code (from git). The programmer believes that obj points to a commit, but this is not checked, either at compile time or at run time.

This paper describes libcrunch, a system which enables *run-time type checking* within unsafe code (focusing on C), while maintaining full binary compatibility (for easy use of libraries), usually requiring no source-level changes, and with low enough overhead to leave "enabled by default" during development. Our research contributions are:

- a novel design based on checking *pointer creation*, not use, and avoiding per-pointer metadata in favour of per-allocation type metadata (§3);
- a C front-end dealing pragmatically with C's untyped heap allocation (§4), common pointer idioms in standard C (§5) and more problematic non-compliant idioms that are nevertheless common (§6);
- an efficient implementation using novel disjoint metadata techniques (§7);
- experimental evidence demonstrating that the system offers good performance, is easy to apply to real codebases, and yields useful feedback to programmers (§8).

2. Run-time errors in unsafe languages

Fig. 1 shows a real code fragment (from git). The programmer believes that obj points to memory holding a commit, but this is not checked, at either compile time or run time.

If the memory holds some other kind of object, a corrupting failure could occur here, likely resulting in a mysterious crash in *later* code, making the cause difficult to debug. In general, for any pointer cast to a type T*—or, perhaps more precisely, for any subsequent *use* of such a pointer—we would like to check that the target storage really does contain a T.

How does this relate to properties checked by memory-correctness tools? In short, memory errors *enable* type errors, by allowing accesses to stray into memory of a possibly different type. The first three kinds of error below are well-understood memory errors, and sophisticated tools exist for checking them [1, 10, 18, 19, 23, 24].

1. **Spatial** memory errors involve the use of an out-of-bounds pointer or array index.
2. **"Too late" temporal** errors involve use of a pointer value that is dangling—examples including use-after-free(), double-free(), use of a pointer passed up the stack, etc..
3. **"Too early" temporal** errors involve use of an unspecified value read from uninitialised storage.

However, these are not the only ways to perform a type-incorrect access in C. The most common other means to such an error are as follows.

4. **Badly-cast pointer** errors are the creation, followed by use, of a pointer of type T* whose referent is not of type T (or compatible).
5. **Vararg** errors involve use of va_arg(_, T) when T is not compatible with the argument passed into the call (if any was passed).

These errors are distinct from the above spatial or temporal memory errors caught by existing tools. The remainder of this paper describes a system which checks for badly-cast pointer errors, and also catches certain cases of vararg errors.

It is also interesting to ask: what other errors, besides the above, can cause type-incorrect accesses in C? We note the following additional classes of error.

6. **Union** errors involve reading from a union member when that union's last-written member is of different type.
7. **Bad-link** errors involve use of a linker-initialized reference whose link-time referent is type-incorrect (e.g. the linker binds a use of a global of type T to a definition of incompatible type U).
8. **Representation-copying** errors include use of memcpy() to copy bytes to a (live, in-bounds) location from where they will be interpreted using the wrong type.

The code in Fig. 2 includes errors of all eight classes. To our knowledge, on the compilers we have tried (gcc and clang), no relevant warnings are generated, nor does

```
int g(float *b, float c, ...);  // declaration                    1
int f(int a) {                                                    2
  int local;                                                      3
  int *p;                                                         4
  {                                                               5
    struct {int x; float y;} z = { 0, 0.0 };                      6
    p = &z.x;                                                     7
    int xx = *(p + 1);          // spatial                        8
  }                                                               9
  int yy = *p;                  // temporal "too late"            10
  int t  = local;               // temporal "too early"           11
  return g((float*) &a, t, 42); // bad pointer cast               12
}                                                                 13
/* --- separate file --- */                                       14
#include <stdarg.h>                                               15
int g(float *b, int c, ...) {                                     16
  union {int x; float y;} z = {.y = 0.0};                         17
  z.x = *b;       // (use of bad-cast pointer)                    18
  va_list v; va_start(v, c);                                      19
  z.x += va_arg(v, float);      // vararg mismatch                20
  z.x += z.y;                   // union error                    21
  z.x += c;                     // exploit bad link               22
  memcpy(b, &c, sizeof (int)); // bad memcpy                      23
  return z.x;                                                     24
}                                                                 25
```

Figure 2. Some even more problematic C code.

any existing dynamic analysis in the literature[1] catch the errors in classes 4–8, although a link-time analysis has been developed for class 7 [2].

In the present paper, our focus will be on classes 4 and (to some extent) 5, neither of which are caught by existing tools. We adopt this focus partly because duplicating existing tools is of limited research interest, but also because, as we will see, a tool focused on these kinds of error turns out to be implementable at much lower run-time overhead than the fairly expensive memory-focused tools. (The same likely holds for classes 6 and 8, but we leave such a tool for future work—except that our tool happens already to catch a small subset of union errors.)

It is worth observing a distinction between *corrupting* and *non-corrupting* errors. In the above code, some errors are non-corrupting: generating a meaningless floating-point value (on line 20), although clearly an error, is not corrupting since the failure is localised to the result value and to any data flowing explicitly outward from it. Bad *pointers* are invariably the source of a corrupting error, because indirect addressing modes are what allow meaningless values to cause corruption in arbitrary program state. (Here we count array indexing as a pointer construct, involving the creation of a transient pointer that is immediately dereferenced.) Since corrupting failures are what make unsafe code especially

[1] Here we are discounting executable C semantics such as CH_2O [14], Cerberus [17] or kcc [8] since none of these currently scales to large codebases. We discuss these systems further in §9.

hard to debug, our emphasis will be on *diagnosing corrupting failures* associated with these errors.[2]

3. Design

Here we set out the core design properties of libcrunch, and discuss some further details of what precisely it checks.

3.1 Goals

Diagnose corrupting failures The primary purpose of libcrunch is to ease debugging the corrupting failures that follow uncaught type errors (not memory errors) in unsafe code. Reporting type errors cleanly, at or near the point where incorrect code is first executed, brings value, by reducing debugging time. This benefit is widely accepted, and we adopt it as a premise of our work. It is precisely the motivation of dynamically type-safe languages, where "safety" is, in the sense of Krishnamurthi and Felleisen [15], the extent to which errors are cleanly trapped. We do not undertake, in this work, to measure this benefit, e.g. by measuring the propensity of programmers to write type-incorrect programs. Instead, our focus is on the design of a pragmatic infrastructure which can bring this benefit to C code (and potentially other unsafe languages too; we discuss this possibility in §7.3). Studies measuring this benefit would certainly be valuable (we are not aware of any) but they are outside the scope of this work.

Help debugging, not (primarily) bug-finding It follows that libcrunch is not a bug-finding tool per se: its express purpose is *not* to detect bugs "lurking" in deployed code, but to assist developers fix errors that are *easily detected* but difficult to diagnose. In other words, they manifest themselves during the usual edit-compile-debug cycle, but are hard to understand the cause of. By contrast, memory errors (both spatial and temporal) tend to "lurk" in codebases, going undetected until after deployment. This is precisely because they are often *non*-corrupting in many builds or executions: an off-by-one bounds error most often reads (or writes) a harmless extra value, while an uninitialised read or use-after-free can do the same. Problems manifest only after deployment, in an unlucky environment that reveals the bug; this motivates bug-finding tools to uncover them sooner. By contrast, the kinds of error we target lead to state corruption much more readily (consider Fig. 1; if the cast is wrong, the called function will immediately consume meaningless data), and are unlikely to "lurk". The difficulty they present is down to the corruption they entail, whose root cause the developer must somehow diagnose.

[2] An interesting distinction is that between *corrupting* failure and merely *distant* failure. Both are difficult to debug, and corruption often causes failure somewhere distant rather than close. However, many distant failures—such as passing a bad value through a long chain of calls, causing an incomprehensible failure in some deeper layer—occur even in *safe* languages. Debugging tools assisting the diagnosis these distant yet *non-corrupting* failures would certainly be useful, in any kind of language. However, this would require different techniques from those we consider here.

Performance By focusing solely on type-correctness, we can produce a low-overhead tool, with around 5–35% slowdown in most cases, and only occasionally higher. By contrast, slowdowns for the fastest precise memory-correctness tools are typically in the range 50–200% (only occasionally lower). This difference owes largely to our maintaining metadata only per allocation, and checking only low-frequency operations such as pointer casts. Unlike memory checkers, the overhead is low enough to be left "on by default" during development, likely only to be disabled in rare "worst case" scenarios. Being turned on by default can clearly help reduce latency in diagnosing certain bugs—those occurring only after some length of time, or nondeterministically—since it avoids the need to re-run the code with the tool enabled.

Compatibility Our system accepts plain C code, and accommodates the overwhelming majority unmodified. libcrunch supports all common C features and idioms, including (but not limited to) casting between pointer and integer, function pointers, pointers to void, multiple indirection, most casts used to simulate subtyping, custom allocators, address-taken stack storage, alloca(), and so on. The guidance required from the programmer is usually limited to a list of allocator and free functions (whether wrappers or nested allocators), simple enough to maintain outside source files. Data representations are unchanged under libcrunch, giving full binary compatibility.

Pragmatism regarding C standards Even within a single source language, such as C, the question of when a check *should* fail is not straightforward. Adhering blindly to a single language specification (say, C11) is not optimal, because much code is not completely compliant, and because hanging to the letter of the standard may bring penalties that outweigh any benefits. Our more pragmatic position, albeit less precise, is to heed what large populations of programs do. As a case in point, memory under libcrunch is given a notion of "type" similar to C11's "effective type" but with some differences. Whereas the instrumentation required to precisely track C11 effective types would be extremely expensive, a much cheaper, slightly stricter approximation nearly always suffices (perhaps after trivial code changes), since most code does not test those corners of the language specification. Conversely, libcrunch allows the user to select more relaxed checking styles when faced with code that is unusually "sloppy", being non-conformant according to the language specification but nevertheless occasionally seen in the wild. We detail these issues in §5 and §6.

Practical usability Widespread use of sloppy idioms in unsafe code mean that false positive warnings occur. Our system strives to exhibit few false positives, but if a check fails, the programmer may allow execution to continue, review the collected warnings much later, and configure suppressions for any false positives. Separately, to avoid the inconvenience of rebuilds, binaries compiled for use with libcrunch

```
if (obj->type == OBJ_COMMIT) {
  if (process_commit(walker, // perform check as side-effect
      ((__is_a(obj, &__typ_commit)) ? (void)0 : report_error(),
        (struct commit *)obj)))
    return -1;
  return 0;
}
```

Figure 3. A sketch (loosely illustrative of our actual automatic instrumentation) of how the code of Fig. 1 might be checked.

```
$ crunchcc -o myprog myprog.c util.c ...
$ ./myprog                          # runs normally
$ LD_PRELOAD=libcrunch.so ./myprog # does checks
...
myprog: Failed check __is_a_internal(0x5a1220, 0x413560
 a.k.a. "uint$32") at 0x40dade, allocation was a heap
 block of int$32 originating at 0x40daa1
...
```

Figure 4. A user session using libcrunch's C front-end.

can still run without it, performing no checks and exhibiting usually negligible slowdown.

3.2 Overview of libcrunch Operation

Fig. 3 shows how our git example might be instrumented to catch a type error at the point of the bad cast, assuming an __is_a() function. Our automatic instrumentation (§7) performs roughly this kind of transformation, and the design and implementation of an appropriate __is_a() function is the main challenge addressed by the remainder of this paper. In contrast to other dynamic checks (SoftBound, Memcheck, Hobbes, etc.), such a function can be implemented using only coarse-grained *per-object* type metadata, without any per-byte, per-word or per-pointer or metadata.

Fig. 4 shows what a user might see at their terminal when compiling and running a mildly buggy program under libcrunch. A compiler wrapper crunchcc performs some pre- and post-processing phases. A key role of these is to gather additional type information that would otherwise be discarded during or after compilation: a source-level pass analyses the use of sizeof around heap allocation calls, and a binary post-processing pass filters and deduplicates the compiler-generated debugging information (DWARF, in our implementation). The output binary is instrumented with our checks, but these have no effect unless the libcrunch runtime is loaded. If the library is loaded, as shown in the figure, execution may produce "check failed" messages as it proceeds, but is otherwise unaffected. By default, execution does not terminate on a failed check (although the ensuing corruption may or may not cause it to crash soon afterwards).

3.3 Pointer Contracts

What makes a pointer bad, and is using one necessarily a "type error"? Informally, we might say that the use of a pointer of type T* is a type error if its referent *is not an instance of* type T. We call such a pointer *contract-violating*. In short, then, our system checks for *creation of contract violating pointers*.

Merely creating a contract-violating pointer is not *always* an error per se, at least in C, so long as that pointer is not used. Nevertheless, correct programs have little reason to create them, so intuitively, the programmer error is more likely to lie at the pointer-creating operation (a bad pointer cast, bad pointer-returning va_arg, bad load of a pointer from a union, or bad pointer arithmetic) than the pointer-using operation (which might be far away, in correct code). One common exception is "one-past" pointers, pointing immediately beyond the end of a buffer; since we do not check spatial memory correctness, this does not concern us (unless such a pointer is later subject to a cast). Since pointer creation is less frequent than pointer use (dereference), checking only creation also promises lower run-time overheads.

The above assumes that all storage has a well-specified run-time type. In the case of C, technicalities of the language hold that this is almost true, but not entirely: storage may have a *declared type* fixed for its lifetime (memory holding local and global variables), or perhaps only an *effective type* that may change (typically heap memory), or perhaps no type (uninitialised heap memory). Roughly, our approach is to tighten this so that all storage has a well-defined type, fixed for its lifetime; we will see that this is not restrictive in practice for most real C code, although occasionally some slight relaxations are necessary. We discuss this further in §5.8. Unused storage pooled by memory allocators is allowed to have no type attached; our run-time explicitly tracks memory allocators (§7.1).

Somewhat more subtly, to accommodate multiple indirection, this "typed storage" property must be hold for *stored pointers* such that these are tagged not simply as "pointer", but as T*, meaning the stored pointer must contractually point to a T. This errs on the side of strictness; again, we will explore the extent to which real C code respects this discipline and ways of relaxing it where necessary.

Our checks are designed to uphold the invariant that no type-incorrect pointer exists—at least *until the first error*. The nature of corrupting failure makes this qualifier necessary, since the first violation is the only one that we can guarantee to catch; afterwards, if we continue execution, all guarantees are off, although in practice, we will likely continue to do useful checks if the program survives. A consequence of this invariant is that there is no need to check pointer dereferences. As long as the invariant holds, any pointer is good to use, by construction.

Since we chose not to duplicate the memory-correctness checks of other tools, our "until the first error" requires additional qualification: until the first reported error *or* the first (unreported) spatial or temporal error. In other words, unchecked memory errors can still violate our invariant, and

our checks preserve the invariant only conditionally: *assuming* no memory errors have occurred, *then* we will trap the first type error. (As a development exercise it will of course be useful to produce a combined tool that checks all of these kinds of error simultaneously. As we discuss later, our implementation already supports certain combinations with SoftBound, Memcheck and similar tools.)

4. From Untyped to Typed Allocations in C

Our "typed storage" assumption would ideally mean that heap allocations are explicitly typed, perhaps occurring in the form allocate(*T*, *n*). By contrast, real C code uses functions such as malloc(*size*). We require some way to map from the latter to the former, so that type metadata is available at run time for each heap allocation.

Firstly, we assume a list of allocation functions, along with their signature. Any argument that is a *type-determining size* must be specially identified. For malloc(sz), this is the unique argument; for calloc(nmemb, sz) this is the second argument; and so on. For user-supplied allocators, the user must supply this list. (A similar requirement is made by other tools that track allocations at run time, such as Valgrind-family tools. We discuss user-supplied allocators shortly.)

Next, to infer the data type being allocated, we pay attention to how the sizes are computed; in C, this means the use of sizeof. In simple cases, for example malloc(n * sizeof (T)), the relevant type is simply the type of the argument to a sizeof operator inside the malloc() call.

In practice, more complex size computations occur. To handle these, we use an intraprocedural flow-insensitive analysis rather like dimensional analysis. An expression sizeof (T) creates a quantity of dimension T. This dimension propagates to other expressions computed from it by arithmetic, according to the natural rules. The dimension which reaches the allocation call's size argument determines the allocation site's data type. Consider the following, which allocates a struct stat.

```
struct stat ss[12];                 // dimensions are:
unsigned n = get_n();               // dimensionless
// ...
long sz1 = sizeof ss;               // struct stat
long m = sz1 / sizeof (struct stat); // dimensionless
size_t totalsz = (n+m) * sz1;       // struct stat
return malloc(totalsz);             // => allocates struct stat
```

The arithmetic done on sizes is usually multiplication by a dimensionless number, to create an array. Addition and division of like-dimension quantities are also common; the former preserves dimension and the latter divides it away. Array dimensions decay to their element type. We permit addition of (dimensionless) padding bytes to a dimensioned quantity, e.g. to include a room for a character string.

Our source-level analysis outputs a classification record for each allocation site per (preprocessed) C source file; this is picked up by our postprocessing step (§7) and propagated to the runtime (§7). Since allocation functions may be called indirectly, any indirect call with a matching signature is considered as a possible allocation site. Since it is the allocation function (callee), not the caller, that is instrumented, spurious matches of indirect call sites are harmless: most are filtered statically by the sizeofness analysis (indirect calls which do not receive a sizeof-derived value are statically ruled out) while, rarely, some result in an extra never-matched entry in the tables loaded at run time (the entry records an indirect call site which happens to receive a sizeof-computed value, but since no allocation function is ever called from that site, the entry is never used at run time).

A final complication comes from expressions such as sizeof (T)+n*sizeof(S). We treat these as effectively synthesising a new struct type, possibly with (at most one) variable-length array encoding the n. Pragmatically, we assume that the textual order in which the components are added up matches their order in memory. (Since addition is commutative, this need not hold, although code which swapped the order would be perverse. We have seen only one such example, in gcc.)

In practice, much real code also defines its own allocators, or its own wrappers for existing allocators. As before, we require the user to identify the existence of these, but code need not be modified.

Besides heap allocations, global- and stack-allocated data also require type metadata. Our implementation handles these cases (see §7).

5. Adding Type Checks to Real C Code

As outlined earlier (§3.3), we are creating a tool that checks for the creation of *contract-violating pointers* by errors *other than* memory errors. The checks we need to insert are therefore the minimum necessary to preserve an invariant: no contract-violating pointer has been created, assuming no prior errors have occurred. "Prior errors" includes both the type errors, which we check (i.e. our checks preserve the invariant up to the first error they catch; after that, checking becomes best-effort since the invariant need no longer hold), and the memory errors, either spatial or temporal, which are unchecked by us (but might be caught by another tool).

5.1 First-order Cases

As we showed in Fig. 3, a cast of the form (T*) p is instrumented to become roughly

```
(__is_a(obj, &__typ_T)) ? (void)0 : __report_error(), p
```

... meaning that a check is added as a side-effect to the pointer creation. The check invokes an __is_a() function and a run-time representation modelling (reifying) type T. The __is_a() function embodies the *contract* (§3.3) that p "really points to a T". This is a well-defined notion since we have type information for all storage.

Our typed heap allocations, together with analogous run-time metadata on stack and static storage, allow us to map any *p* to a *containing allocation* and a type for that allocation.

(We describe the implementation of this mapping in §7.) The logic of __is_a() is then to enumerate the objects beginning at p, which is some known offset from the allocation start. Consider the following example.

```
struct ellipse {
  double maj, min;
  struct point { double x, y; } ctr;
};
struct ellipse *e = malloc(sizeof(struct ellipse));
double *d = (double*) &e->ctr;
```

The cast to double* of the ellipse's centre proceeds first by looking up the type of the entire malloc'd heap block allocation, then searching for a double object starting at the relevant offset (i.e. that of &e->ctr from e). Since objects nest within one another—here doubles and point within an ellipse—there may be many objects starting at this offset. If one is of type T, the created pointer satisfies its contract; otherwise, the check fails.

As discussed earlier (§3), we chose to assume spatial and temporal memory correctness (already checked by other tools), so we need not instrument pointer dereferencing, arithmetic or indexing. The same assumption removes the need to worry about temporal use-after-free errors (formerly type-correct pointers now pointing to freed-and-reused memory) or uninitialised reads (including those reading uninitialized pointer values). Non-pointer-yielding type errors (casts or va_arg whose result is not a pointer) do not cause corrupting failure, so are not checked. We must check all casts whose target type is a non-void pointer, and also pointer-yielding use of va_arg. C performs *implicit* downcasts from void*; we check these as if an explicit cast were present.

When checking a pointer cast, note that the cast-from type is irrelevant; the cast-to type and the type of the underlying object are what matters. Casts to void* need no check. We also do not check casts to char*, for two reasons. Firstly, char is ambiguous: it is the data type used both for character data *and* for opaque, uninterpreted memory; checking would create false positives in the latter usage. Secondly, char is not amenable to allocation site analysis (§4): since it is known to have size 1, sizeof (char) is generally omitted. Consequently, libcrunch considers char to be the type of untyped, unclassified memory. This omission is a pragmatic one. It is unfortunate that C does not provide a distinct data type for uninterpreted bytes, since this would allow us to perform additional checks, but the change would be highly disruptive. (We could check all writes through char* and forbid those that touch typed memory, but this would be expensive—e.g. most string manipulations would trigger one check per character.)

Even if all pointer casts check successfully, how can we be sure that the *contents* of memory are correct, when they are themselves pointers? Consider the following, where field t has type T*; we load this pointer via another pointer, s.

```
S *s = (S *) foo();
T *t = s->t; // does t really point to a T?
```

After loading t, we do not need to check that it points to a T (or null) because, by our invariant, when t was last written, all pointers in circulation were type-correct—including the expression written into field t at that time (and, more subtly, any pointer expression that was used to calculate the address to write to). In short, the storage contract is enforced (modulo memory safety) by the combination of the C compiler (statically insisting on casts) and our checks (dynamically checking those casts).

What we have just seen is a first-order case: the S* and T* are singly-indirected data. Some *higher-order* cases, involving functions and/or multiple indirection, are more subtle and require us to refine both our checks and our invariant.

5.2 Example: Callbacks and void Pointers

Consider a common but more complex idiom in C.

```
/* library provides... */
typedef (*cb_ptr_t)(int value, void *cb_arg);
void call_for_each_num(cb_ptr_t cb, void *arg);

/* client declarations */
struct my_cb_args {
    // ...
} args = { ... };
int my_cb(int value, void *my_arg);

/* client code */
void f(void) {
  // ...
  call_for_each(my_cb, &my_cb_args);
}
int my_cb(int value, void *cb_arg) {
    struct my_cb_args *my_real_args
     = (struct my_cb_args *) cb_arg;
    // ...
}
```

This is a classic use of pointers to void: the library supplies generic callback dispatch code, which invokes a user-supplied callback accepting a user-supplied argument. The user argument is packaged as a pointer to void, so the library code remains generic. It can point at any object the user chooses, and the cast back occurs in the user's callback. Our system naturally accommodates this. As C demands, conversion of any pointer to void* is permitted. At the point of the cast back to my_cb_args*, an __is_a() check will occur on cb_arg, dynamically looking up the type of the object pointed to by cb_arg and verifying that it is, or contains at offset zero, a my_real_args instance. It does, so the check succeeds. If the user had passed a pointer to an object of some other type, this would be caught at the point of the cast, rather than proceeding with a type-incorrect view of memory and potentially corrupting the program state.

5.3 Multiple Indirection

Perhaps surprisingly, multiply-indirect pointers typically require no special treatment. Consider the following.

```
void *p = /* ... */; // acquire some pointer, as void*, but...
*(int**)p = &argc; // it points to a pointer−to−int
```

The cast succeeds if and only if the memory pointed at by `p` is an `int*` according to its allocated data type. This is simply the usual behaviour of `__is_a()`. It is also exactly what the contract of `int*` demands: users of `*p` may assume, without check, that this memory holds a pointer to an `int`.

Later we discuss some more subtle cases of common idioms that are not compliant C (§6).

5.4 Function Pointers

Casts of function pointers raise analogous problems to multiply-indirect pointers: they risk creating a capability on which future errors might go unchecked. For example, if we cast a `void(*)(float*)` to a `void(*)(int*)`, we can now pass a pointer to `int` to a function that expects a pointer to `float`, with no check in place to catch this. To `libcrunch`, a function is an object like any other, with a reified function type. The existing `__is_a()` test can cope with function pointers, and will correctly fail the cast in this example. In fact, it fails any cast to a function pointer type which doesn't exactly match the signature of the pointed-to function.

Standard C is strict about function pointers: the only casts that are portably supported are casting to `void*`, and casting from `void*` back to the original type. Nevertheless, we can support some more sloppy code fairly easily, as we detail later (§6.6).

5.5 Simulated Subtyping via Contained Structs

The usual approach to simulate `B` being a subtype of `A` is containment: `B` contains an `A` at offset zero. This is handled directly by `__is_a()` as previously described.

5.6 Unions

We noted that `unions` and `va_arg` allow type-unsafety without pointer casts. Our system includes only limited checking in these cases.

For `unions`, we can distinguish two cases: where all members of the union are simultaneously valid (e.g. to allow viewing the bits of a floating-point number as an integer) or when only the last-written member is meaningful. For non-simultaneous `unions`, precise checking would mean tracking which member of the union had last been written, and checking that reads access only that member. We have yet to add this; currently we assume all `unions` are simultaneous.

Despite this omission, some checking is performed on `unions`. For an arbitrary type t and a pointer p pointing at a union object, `__is_a(`p`,` t`)` checks that *some* member of the union conforms to t—i.e. the cast is sane, although not necessarily correct with respect to the last-written member.

The hardest cases involve code which takes the address of a member of some union type `U`. Suppose the member has type `T`; we now have a `T*` which is indistinguishable from other pointers to (non-union) `T`. We leave treatment of this to future work—likely it could be handled with some combination of an intra-file interprocedural analysis (to catch "transient" non-escaping cases) and a "trap pointer" representation (issuing a fake pointer via which accesses can be trapped and emulated with appropriate checks).

5.7 Variadic Functions

For `va_arg` calls yielding a pointer, we check the result as if a cast were present. This is sufficient to uphold our "no bad pointers" invariant. Other `va_arg` errors, like reading a `float` when an `int` was passed, are not caught. Doing so would be feasible, using a shadow stack to track the arguments passed at variadic call sites.

5.8 Relationship to Effective Types in C

Memory in standard C has a notion of "effective type". This was introduced to forbid aliasing in places where it would prevent optimisation. Storage may have a *declared type* fixed for its lifetime—this is memory holding local and global variables, whose type is "declared" within the language. Heap storage, returned by library calls, has at most an "effective type"; unlike declared types, effective types may be changed simply by *writing* or using `memcpy()` to copy data into it from an object of another type. (This is the device in C that allows a heap object to legally "take on" the type by which it is used, after having been created by an untyped primitive such as `malloc()`.)

In `libcrunch` we have a slightly different, tighter scheme: the `malloc()` itself is typed. This is usually a better model of programmer intent, since, as we noted in §4, programmers necessarily have some data type in mind when they size a heap allocation. This tightening also avoids the need to instrument writes, greatly lowering run-time overheads; if we were to hang to the letter of C11, trapping writes would be necessary for catching changes to the effective type.

The "type changing" ability of writes has surprising consequences for unwary C programmers. For example, it means that some natural refactorings, such as making a previously heap-allocated buffer be instead allocated statically, may be illegal (if writes to the buffer disagree with its now-declared type). Programmers rarely rely on type-changing writes; rather, operations which change the type tend to be procedurally abstracted in some fashion, typically as `realloc()` or `memcpy()` calls. This means they could be trapped efficiently; an obvious extension of our system is to check `memcpy()`, although currently it does not.

In the next section, we will see one case where a slight refinement to our approach is necessary to accommodate some real (albeit not-strictly-conforming) C code.

6. Accommodating Sloppy C

Some C code is formally undefined, therefore buggy, but nevertheless occurs in the wild. A common giveaway is when code requires compiler options to relax the language spec; in this section all the sloppinesses we consider require the -fno-strict-aliasing option (on GNU, LLVM, and Intel compilers).

6.1 Simulated Subtyping via Structure Prefixing

Some data types simulate subtyping more implicitly than the usual structure containment (§5.5). A common pattern is to define A as a fieldwise prefix of B, (rather than B containing an A). To handle this, libcrunch reads an environment variable which requests that casts to [pointers to] the listed target types use the more relaxed __like_a(p, t) check. This treats a structured type t as a pattern rather than a single type. We unwrap t into its n constituent fields, say of types s_i, then check that the memory at p consists of a sequence of subobjects q_i, spanning the length of t and each satisfying __is_a(q_i, s_i). Note that the check remains relatively strict, because we unwrap t only one level, not necessarily down to primitive types as in physical typing [4]. (We have not seen any real code which requires the full permissiveness of physical typing.)

6.2 Simulated Subtyping via Padding

The Berkeley sockets API's sockaddr exhibits another pattern: it includes padding bytes which "subtypes" (like sockaddr_in) "fill in" without changing the structure's length. We extend __like_a() such that any char array fields in t are treated as padding and match a sequence of arbitrary subobjects of the appropriate length.

6.3 Mixing Signed and Unsigned

We overload __like_a for integer types to mean a signedness-oblivious match, since some C code is sloppy in this regard. As it happens, the C standard does allow this, whereas by default libcrunch does not, since in our experience it is more likely to be an error than legitimate. Users who find otherwise may add integer types to the "like-a" list.

6.4 Mixing void* and Other Pointer Types

Our treatment of multiple indirection can in principle be defeated by code which weakens (to void) the target type of a pointer. Consider the following.

```
struct foo;
void get_foo_ptr(struct foo **fp);
// ...
void *p; // client only uses it opaquely, so declare a weak type
get_foo_ptr((struct foo **) &p); // cast fails __is_a!
```

The cast is flagged as an error because the stored pointer's actual contract is void; it is not safe to later cast it to something stronger (since a void pointer holds an arbitrary address, which e.g. need not be safe to double-dereference). In general this practice is not legal in C, because the representation of void* is not guaranteed to be identical, or even the same size, as that of unrelated pointer types [11, §6.2.5 pt. 28].

Note that this is an inversion of the usual use of void*, in which the actual storage's contract is *stronger* than void, and the void* view is a temporary weakening that is strengthened again by a later cast. This is a synthetic example, and we have not seen this problem in the wild; if it did occur, the programmer would likely either configure a suppression of the warning, or fix the code by strengthening the type of p.

6.5 Parametric-style Polymorphism

Some *polymorphic* C code demands greater freedom with multiple indirection, to store pointers which violate their strict storage contract. Consider the following (from tcc, the Tiny C Compiler[3]).

```
void dynarray_add(void ***ptab, int *nb_ptr, void *data)
{   int nb, nb_alloc;
    void **pp;
    nb = *nb_ptr;
    pp = *ptab;
    /* every power of two we double array size */
    if ((nb & (nb - 1)) == 0) {
        if (!nb) nb_alloc = 1; else nb_alloc = nb * 2;
        pp = tcc_realloc(pp, nb_alloc * sizeof(void *));
        *ptab = pp;
    }
    pp[nb++] = data;
    *nb_ptr = nb;
}
Sym *sym_pool;
/* ... */
dynarray_add((void ***) &sym_pools, &nb_sym_pools, sym_pool);
```

The function creates or resizes a vector of pointers. The client holds a pointer to the vector with an accurate type (here the vector is of Sym*) while the generic code uses void*. As before, this practice is not legal in C. Having compatible representations is a property commonly documented by implementations, however. To accommodate this idiom, we relax the contract enforced for *pointers to generic pointers*[4] (GPPs) and compensate by adding extra checks around these pointers. Any GPP has a *degree* of at least 2 (e.g. void**). GPPs of degree n may point to *any* pointer of degree $n-1$ or higher (even to non-generic pointers, but never to non-pointers). This is sufficient to ensure we never *read* a bad pointer through a GPP; it is unsafe for writes, since given a void** we could write *any* void*, whereas the target storage might be contractually obliged to store T*s, say. We there additionally check writes made *through* GPPs, to enforce the contract of the target storage. Consider writing a void* through a void**. From the degree constraint, the target storage must be a T* for some T. We discover T dynamically by inspecting the metadata for the target storage. We then dynamically check that the written value is indeed the address of a T, hence accommodating the poly-

[3] http://bellard.org/tcc

[4] Note that ordinary generic pointers, a.k.a. void*, are not affected.

morphism dynamically. Functions __is_pointer_of_degree() and __can_point_to() implement this.

When objects of generic pointer type are heap-allocated, like the array of void* in the example, their storage contract is recorded to be "loosely" void* . The contract is made strict by the first cast producing a non-generic pointer to the object. After a cast to Sym**, say, the contract would be Sym*. This can be said to "instantiate" the void.[5]

Note that this is the relaxation of our "decide type at allocation time" property anticipated earlier (§5.8). Note also that it still stops short of trapping most writes. The arrangement is closely analogous to the compromise of arrays in Java: the array subtyping rules are such that in exchange for reads from an array to be cheap (unchecked), writes must be checked to ensure that they preserve the contract of the underlying array (which might be of some more specific type than the local array reference's own type), raising an exception if not.

It is worth reiterating that this relaxation only comes into play with *multiple indirection* of void. Most polymorphic C code, including "simulated object-oriented" styles, gets by with simple pointers-to-void, which are not affected: one can never read or write *through* a pointer-to-void, so any dereference must be preceded by a cast, which is checked in the usual way. Multiple indirection of existential void most commonly occurs with generic containers containing pointers, where in the generic code, a pointer to void* is standing in for some actual pointer to T* (not just a pointer to T!), and the code is allocating space for pointers on behalf of the user.

6.6 Sloppy Function Pointer Casts

Earlier (§5.4) we identified "exact" function pointer casts as requiring no special treatment from libcrunch.

Some casts to function pointer types are safe even though they are inexact: these are those that cast to a supertype signature, according to the usual function supertyping rule: we can narrow the arguments of type T* to S* if S is a subtype of T (every S *is a* T), and, conversely, widen the return type. To allow code performing these always-safe casts, we created a specialised check function that permits cast-to-supertype, __is_a_function_refining(), used for all casts to function pointer type, rather than the usual __is_a().

What about casts to non-supertypes? The only common usage we are aware of is with callback signatures, where a sloppy programmer might cast the address of int cb(T* arg) to int(*)(void*). This cast is *not* safe, because the new signature permits a superset of the allowable arguments (a caller may pass any pointer, not just any pointer to FILE as the function requires). Usually the desirable thing here is to fix the code to define int cb(void *arg) and perform the cast inside the callback; this also renders it standard-compliant C. Nevertheless, to accommodate this slop we also provide a check-on-use mode for function pointers, enabled by an environment variable. This performs checks on every pointer argument at every indirect call site, so can introduce noticeable slowdowns in some code; it is disabled by default.

Our implementation currently fails (dynamically) all casts to variadic function pointer types. These are very rare. "Exact match" cases, say from void* to a function pointer type exactly matching the target function's variadic signature, could be supported easily. More complex cases could be handled using the same shadow stack we mooted earlier for va_arg (§5.7), by generating a "checked" wrapper function which pops the variadic argument range passed from the caller (using the same va_arg check) and checks against the signature of the underlying callee. Note that if the callee is itself variadic and has been instrumented with our checks, it will do its own checks for the relevant portion of its argument list, but this need not correspond to the portion checked by the wrapper, which is determined by the *cast-to* type's variadic signature. (Some care is required to preserve function pointer equality under this wrapper approach.)

In older C code, a similar but distinct kind of cast is sometimes seen: a cast to a function pointer type *without arguments*. For example, one can write (int(*)()) printf to erase argument types from view. This is subtly different from a variadic signature: whereas a variadic function signature gives one or more arguments (such as printf()'s format string) whose values will be used to decode the remainder of the list, here no argument information is retained at all. Producing a binary-compatible wrapper function at the point of such a cast is potentially difficult, since nothing is known about what or how arguments will later be passed. This is an outdated C idiom, and code using it is fairly rare. Safely permitting such casts could no doubt be achieved by instrumenting instead the *use* of function lvalues that have argumentless types. Currently we do not implement this; instead, casts to such types provoke a warning at run time.

6.7 Sloppiness and False Negatives

The instrumentation done by libcrunch is comprehensive, meaning that every possibly-failing pointer cast that is seen at compile time is instrumented with a check. Without sloppiness, these checks are precise, in that they cannot pass by chance: checks pass only if the allocation metadata precisely agrees with the cast-to type. This contrasts with some other styles of checking, which could be called "best-effort", such as the "redzones" used for spatial memory checking in ASan [23] and similar bounds checkers. These will only reliably catch a small overrun (up to the redzone size); they have some chance of catching a large overrun (if it lands in an unused piece of memory, say) but will not catch a large overrun that happens to land inside a valid object.

[5] In other systems, like Cyclone, the analogous problem is solved by *statically* instantiating these "existential voids", i.e. by rewriting dynarray_add() and similar functions to use a parameterised type T and checking that it is used consistently. This approach requires programmer intervention in general, so is not source-compatible.

In a sense, then, given that libcrunch's approach is comprehensive and precise, false negatives are absent by construction. However, this only holds if one does not use sloppy features. The purpose of enabling sloppy treatment is to suppress warnings which the programmer considers false positives; naturally, if such features are enabled inappropriately, false negatives become possible. Since sloppy checking must generally be turned on in a fine-grained manner (once per cast-to data type), this is unlikely to happen accidentally. The most likely accident comes from over-sloppy treatment of signed versus unsigned: perhaps only a small amount of code intends to treat these sloppily, whereas turning on sloppy treatment program-wide may mask genuine bugs in other places. Our implementation, described in the next section, assists with this by making it easy to turn on sloppiness on a file-by-file basis. Since sloppiness settings are controlled by environment variables, file-by-file variation can conveniently be coded up in a makefile, not unlike the usage of variables such as CFLAGS.

(Also, of course, this discussion is neglecting that in any given application of libcrunch there will be some uninstrumented code: inline assembly, certain library code, etc.. There may also be *allocations* for which metadata is not available, even when the code using those allocations *is* instrumented; this can happen when memory allocated by an uninstrumented library is used by instrumented code. At run time, liballocs prints a count of queries that failed in this way, but no further warning is printed by libcrunch when this happens. In practice this is a further source of false negatives—albeit often with the "easy" solution of recompiling the uninstrumented libraries.)

7. Implementation

Our implementation consists of compile-time passes to extract allocation-site metadata and perform instrumentation, a C compiler wrapper (crunchcc) which invokes these, an experimental C++ compiler wrapper (not described), and the libcrunch runtime.

Our implementation currently supports only the x86-64 GNU/Linux platform, but presents no particular obstacle to ports to other platforms. Our experiments have largely used gcc as the underlying C compiler, but nothing precludes the use of other compilers such as Clang/LLVM.[6]

Source-level passes Both the source-level sizeof analysis and the necessary instrumentation of C code (pointer casts, va_arg, etc.) are implemented using CIL [20], an OCaml library for analysis and instrumentation using a simplified C-like abstract syntax.[7]

[6] In fact, a Clang-based front-end to libcrunch (replacing the CIL-based instrumentation) already exists, as described by [7].

[7] At the time of writing, CIL continues to be maintained (by Gabriel Kerneis). Various patches originating from libcrunch development have been accepted upstream.

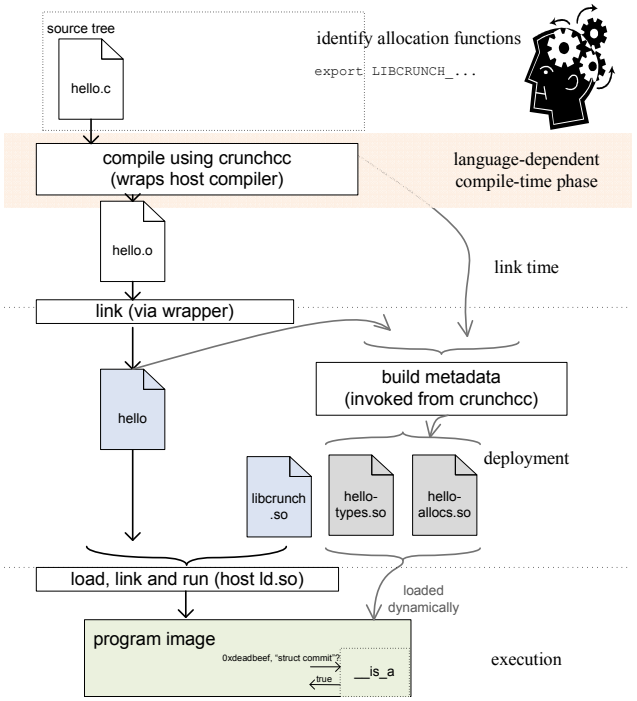

Figure 5. How libcrunch works: extra metadata originating in source code is threaded through an otherwise conventional toolchain with added compile-time instrumentation.

Compiler wrapper CIL's cilly compiler wrapper is used to drive compilation. However, this is augmented with additional pre- and post-passes, described shortly.

Runtime The libcrunch runtime is implemented as an extension to liballocs [13], a run-time system that augments Unix processes with a reflective metamodel based on the abstraction of *typed allocations*. The model of liballocs is an excellent fit for our typed storage model, and provides a large amount of ready-made infrastructure for mapping between addresses and type information. It also allows us to support user-supplied allocators (a frequent omission in similar tools) with no special support or source-level changes. (By contrast, for example, Valgrind-family tools currently require source-level annotation of nested allocators.)

7.1 Background: liballocs

Our implementation of libcrunch is built on several key services provided by liballocs

Reified data types Under liballocs, data types are reified at run time. DWARF debugging information provides the basic notion of data type, but link-time tools (see below) postprocess these so that each type has an efficient in-memory representation called a *uniqtype*. Notably, these are made *unique* at run time, such that only a single copy of a given datatype exists throughout the whole process, hence allowing an "exact type" test to be a simple pointer comparison. Reified

types have symbol names visible to the linker, and the standard link-time technique is used to keep them unique (global symbols and section groups, on ELF platforms). Symbol names also include a hash code which accounts for substructure (length and encoding of primitive types; members, including names, for composite types), avoiding name collisions in the case of distinct like-named data types. Typedefs and other type synonyms (`signed` versus `int`, say) are encoding using aliased linker symbols. Stack frame layouts are unified with data types; they are described much like C `struct` types.

Allocation tree The `liballocs` runtime infers and dynamically maintains an *allocation tree* at run time, reflecting how user-supplied allocators may nest allocations one inside another. For example, when a user-declared allocation function returns a pointer inside a previously `malloc()`-issued chunk, `liballocs` understands this as a nested allocation, where the user allocator is parcelling out a slab of memory that it obtained in from `malloc()`. Queries such as "what is the type of `*p`?" are dispatched to the allocator managing the *leaf* allocation spanning address `p`. A special allocator module is assigned to stack regions; it implements the same interface, answering queries by walking the stack and looking up the uniqtype for the stack frame spanning the query address. (Since stack storage comes and goes without notice, internally within `liballocs` less caching of stack walk results is possible, compared to heap queries which may be cached and later invalidated on deallocation. This means that programs making unusually heavy use of the stack run unusually slowly under `libcrunch`, as we will see later.)

Pluggable allocators Allocators are explicitly represented at run time, as entities managed by `liballocs`. Built-in allocators represent mechanisms provided by the system (`mmap()`, `sbrk()`, kernel stack allocation, and "static" allocation usually performed by the dynamic loader), and by the language library (`malloc()`). User-supplied allocators are declared via environment variables and are instantiated similarly by `liballocs`. Each allocator provides its own implementation of a common metadata query interface, including `get_type()`, `get_base()`, `get_size()` and similar operations, implemented using whatever data structures the allocator chooses (bitmaps, allocation header words, etc.). The combination of per-allocator metadata queries and the allocator tree brings a *whole-address-space metadata query interface*—in principle, any valid pointer can be presented to the query interface, yielding metadata on the object it points to.

Toolchain wrappers The `liballocs` distribution includes compiler wrappers and related tools for gathering type information and allocation-site information (the addresses and source coordinates of calls to allocator functions/wrappers). These tools postprocess compiler-generated debugging information, generating a shared object collecting together all the DWARF-derived uniqtype definitions for the linked binary. They also collect *allocation site* tables used by the static, stack and heap allocators' index structures, allowing mapping of (respectively) static addresses to their types, code addresses to their stack frame's "type" (layout) at that point, and *allocation site addresses* to the type that they allocate. These are loaded at run time by `liballocs`.

7.2 Adding Support Within `liballocs`

Our runtime implementation has contributed certain features and optimisations to `liballocs` that benefit `libcrunch`.

Reworked top-level data structure As described in the literature, `liballocs` provides a "mapping table" tracking the top-level memory allocations, a.k.a. memory mappings, in the process. To allow efficient queries on arbitrary addresses, `liballocs` maintains an *unreserved* virtual memory region containing a two-byte integer for every page in the address space; the two bytes, if non-zero, are an offset into a (fixed-size, preallocated) "big allocation" table. "Unreserved" means the operating system does not reserve space for the (huge) unused portions. Previously this region was only used to track a flat collection of memory mappings. Since `libcrunch` issues very frequent queries on allocations nested some way beneath the top level (e.g. `malloc()` chunks, stack frames, etc.). We reworked this part of `liballocs` so that it instead tracks a hierarchy of "big allocations", including not only memory mappings but also any sufficiently large chunk and any chunks parcelled out by nested allocators . This allows short-circuiting metadata lookups from a virtual address (page number) down to somewhere near the leaf of the allocator tree, contributing to the run-time efficiency of `libcrunch`.

Indexes in `liballocs` In `liballocs`, structures holding disjoint metadata are called *indexes*; an index accepts queries for arbitrary addresses in the allocated region and returns the corresponding allocation site and/or other metadata. Note that the queried address need not be the allocation start address; queries are effectively *range queries*, querying the "nearest preceding" allocation start. In the case of ready-made index implementations, link- and load-time interposition is used to generate upcalls on each allocator invocation. (Other allocators are free to use other mechanisms to implement the query interface. For example, a high-performance `liballocs`-aware garbage-collected heap would probably use header words maintained directly within the inlined allocation path, rather than upcalls and a disjoint index.)

Indexing the `malloc()` heap Specialised per-allocator index implementations are an obvious way to improve `liballocs`'s performance. The index implementaton for the `malloc()` heap is particularly important to `libcrunch`. Chunks in the `malloc()` heap range in size from a few bytes to gigabytes. The largest chunks are already adequately handled by `liballocs`'s "big allocation" mechanism (their metadata is available direct from the big allocation table). However, smaller

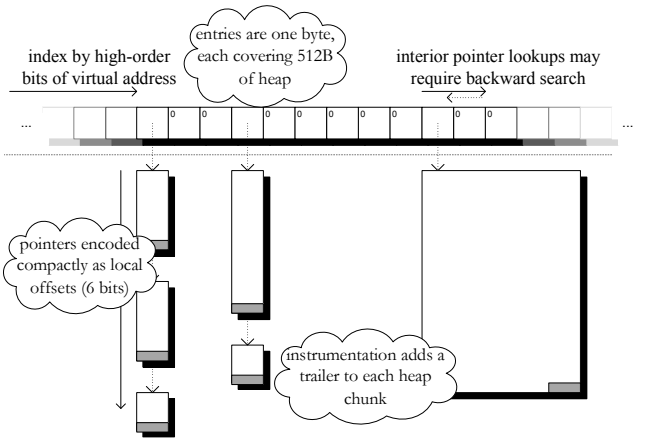

Figure 6. The malloc() index's heap-threaded memtable.

chunks are common, and require careful treatment. The generic index implementation in liballocs, ("generic small"), although flexible, is compromised by its decision to cater to all allocators, even those differing substantially from malloc()—in particular, allocators which allocate tightly-packed and/or fixed-size entries. To improve on this for the case of malloc()-style heaps, libcrunch includes a specialised index which exploits the following properties of the conventional malloc() interface:

- an arbitrary "allocation size" argument (which our instrumentation is free to increment);
- an 8- or 16-byte minimum alignment (i.e. limiting the overall allocation density or "object pitch");
- ability to query a chunk for its usable size (malloc_usable_size() call);
- an "mmap() threshold" for size, allocations larger than which are indexed elsewhere (noting that we instrument mmap() also).

Our approach combines three ideas.

Local metadata We store metadata directly in the allocated chunk, by incrementing the size at allocation time. This adds a one-word overhead per heap chunk. (Of course, in the worst case, alignment constraints might increase this to two words minus one byte.)

Link chunks into buckets We also increment the size to allow room for a doubly-linked "bucket" list threading together nearby chunks. Since all chunks are aligned to 8 or 16 bytes, and only nearby chunks go in the list, only a few bits are needed for each pointer. In our implementation, chunks start within the same 512 (2^9) bytes of virtual address space and are aligned to at least 8 (2^3) bytes, so 6 bits suffice.

Index buckets using a virtual region As in the "big allocation" index, we provide a top-level lookup structure using a large linear region of uncommitted virtual memory. Each bucket list "hangs" off an entry in this region. Each entry is simply a pointer to the chunk at the head of the list. Like the linked-list pointers, only a few bits are needed; the high-order bits of the chunk addresses are implied by the offset at which the list "hangs" in the linear region.

The resulting structure, shown in Fig. 6, which we call a "threaded memtable", looks rather like a hash table. Crucially, however, it preserves locality (cf. hashing the address, which does not), hence allows the range queries we require. Queries on interior pointers require a backward search for the preceding chunk start. By knowing or querying the threshold above which the malloc() implementation uses mmap() (e.g. 128kB), we can bound the backward search to a relatively short distance (e.g. 256 buckets, if each covers 512 bytes). Buckets tend to be small: a 512-byte bucket will hold at most 32 16-byte-aligned chunks, and typically far fewer (folklore holds that chunk sizes typically average around 80 bytes).

7.3 Other Implementation Concerns

Languages other than C Although our instrumentation logic is highly specific to C, our runtime is largely language-agnostic. Even at present, linking in certain amounts of non-C code works well with libcrunch. In particular, although that code's operation is not checked, there is no problem with *loss of pointer metadata* (a problem suffered by some systems when pointers are passed to or from uninstrumented code) because the metadata in libcrunch is per-allocation. Moreover, liballocs can already trap and (in some cases) attach metadata to allocations made from other languages such as Fortran, C++, etc..

Combining with other tools Since we avoid duplicating memory correctness checking, it is useful to combine libcrunch simultaneously with other tools. This already works fairly well. For example, following addition of type checks by crunchcc, we can successfully compile the resulting C code using SoftBound, gaining spatial memory checks. Currently, the main limiting factor is the use of virtual memory in the libcrunch runtime. Since the runtime makes adventurous use of the virtual address space, it precludes naïve word-by-word shadowing—attempting eagerly to shadow libcrunch's own metadata structures quickly exhausts memory. This problem affects SoftBound's fastest "shadow space" metadata implementation, so we instead use the trie-based SoftBound runtime included in the public release. The same problem defeats Memcheck's shadow memory technique. However, it remains possible to run Memcheck on a libcrunch-enabled binary, simply by omitting to preload libcrunch (recall Fig. 4); of course, no type checks occur, but there is no need to rebuild the code. We hope to improve on this in future by producing a generalised shadow-memory runtime within liballocs, based on the techniques of Zhao et al. [28].

API and other applications As so far described, the only clients of the libcrunch and liballocs APIs are the instrumentation which our front-end inserts. However it is perfectly possible, and often useful, to code against these APIs directly, after explicitly including libcrunch's and/or liballocs's header files. For example, one can make assertions about the type of storage on the end of a pointer, or about its allocation site. It is also possible to invoke the relevant routines from the debugger, to interactively query these properties.

Status As described so far, libcrunch is implemented, working and stable at the time of writing, and supports multithreaded code. Source code is available online.[8]

8. Evaluation

This section will demonstrate that:

- libcrunch stays acceptably quiet on most well-tested code (false positives);
- libcrunch successfully catches the kinds of violations it was intended to (true positives).
- libcrunch's run-time overhead is generally low enough to leave it enabled during the course of development;

Anecdotally it is hard to find a C programmer who cannot recall plenty of mysterious crashes whose cause turned out to be a bad pointer cast. However, as noted earlier (§3) we do not, in this work, attempt to turn these anecdotes into evidence—we do not attempt to measure the propensity of C programmers to write bad pointer casts, or to spend time debugging them.

8.1 Studying False Positives

Since we wish to study both performance and false positives, we considered all SPEC CPU2006 benchmarks written entirely in C. We exclude libquantum since it uses C99's _Complex which is unsupported by CIL. We were forced to exclude perlbench since our system currently fails to deal with Perl's (highly idiosyncratic, nonconformant) code (see Appendix B). The remaining ten benchmarks compiled, ran and passed output validation under libcrunch. Each consists of well-tested code from a real application.

Method For each codebase we built and applied any compile fixes (required by CIL or, less commonly, by the underlying gcc compiler). Then we did an initial libcrunch-enabled build whose purpose is to elicit and impart "developer knowledge" of the code to libcrunch. As noted earlier (§7.1, §6) both liballocs and libcrunch accept hints via environment variables, informing them of details such as which functions are allocation functions and which data types might require special treatment (such as __like_a() rather than __is_a() checking). Setting these environment variables requires a process of manual code inspection, guided by diagnostic messages from an initial run under libcrunch. If we were the original developers of the code, writing this information down would be substantially easier; to help, liballocs dumps a list of unrecognised allocation sites, since the most common requirement is to set environment variables naming allocator wrapper functions. Similarly, libcrunch warnings reporting similar-named types are a giveaway of likely __like_a() cases. Once the environment variables are set correctly, we re-build to effect any changes to the instrumentation. At this point, we have done as much as we can to impart developer knowledge of the code to libcrunch, and so any remaining libcrunch warnings may be classified as true or false positives. In the following paragraphs we describe our experiences with the SPEC CPU2006 codebases, which are functionally well-tested, so more likely to exhibit false positives than true positives. We summarise our findings in Table 1; as we will see, it turns out useful to distinguish "unhelpful" false positives (outright failure of the tool) from "helpful" ones (genuine code quality problems, albeit not strictly bugs). After counting these, we apply any obvious fixes to the code, with the goal of reducing its warning count to zero (or as close to zero as possible); we summarise these changes in Appendix A.

Straightforward cases The **gobmk** (Go-playing), **hmmer** (gene-searching) and **sjeng** (chess-playing) benchmarks presented no problems. The latter two perform very few casts. The **mcf** combinatorial optimisation system proved a good test for our improved memory index (§7.2) by allocating huge arrays, and correctly reported no errors. The **milc** quantum chromodynamics simulation performs non-trivial casts in a tight loop, so was a performance challenge, but ran without false positives. The **sphinx3** speech recognition system involved complex custom allocators, some use of pointers to generic pointers (§6.5) and a checksum function that uses signed integers as if unsigned (§6.3). Without the signedness relaxation, libcrunch correctly flags the latter. The relevant environment variable avoids these warnings.

h264ref After two one-line compile fixes, this video encoder reported 27 failed casts. We tracked it down to two copies of the following code.

```
if(((*array4D) = (short****)calloc(idx,sizeof(short**))) == NULL)
  no_mem_exit("get_mem4Dshort:␣array4D");
```

The allocation is sized using short**, but clearly this should be sizeof(short***). The sizes are (usually) the same, so conventional toolchains do not expose this error. This is a real defect, if not (behaviourally) a bug per se; we count it a "helpful" false positive.

lbm This computational fluid dynamics system allocates a very large array (or "grid") of double. Initially we saw many millions of failing casts to unsigned* on this grid. On investigation, the code was treating every 20th double as an array of bit-flags: the storage is allocated as double but is

[8] http://github.com/stephenrkell

always read and written as unsigned. The following macros were used for this.

```
#define MAGIC_CAST(v) ((unsigned int*) ((void*) (&(v))))
#define FLAG_VAR(v) unsigned int* const _aux_ = MAGIC_CAST(v)
// ...
#define TEST_FLAG(g,x,y,z,f) \
   ((*MAGIC_CAST(GRID_ENTRY(g, x, y, z, FLAGS))) & (f))
#define SET_FLAG(g,x,y,z,f) \
{FLAG_VAR(GRID_ENTRY(g, x, y, z, FLAGS)); (*_aux_) |= (f);}
```

We worked around this problem by a small change: allocate the grid as a simultaneous union (§5.6) of double and unsigned. The change can be made at effectively a single point in the code (one C file and its header). Of course this change is overly permissive, in that it allows *all* of the double values to be treated as unsigned; a better solution would to introduce a more abstract data type. Effectively, this warning resulted from the programmer "opting out" of data abstraction. Although not a bug *per se*, it was appropriate for libcrunch to flag it, so we count it as a helpful false positive. It is also a rare instance of C's "effective type" rules being exploited: since the array is allocated on the heap, it is permitted to use a differently-typed write to change the effective type (here from double to unsigned). The same would not be permitted for a static or stack-allocated array.

bzip2 A few casts in bzip2 are used to set up differently-typed pointers to the same working array, such that different subsets of the array are used to hold different-width integers at different times. As one would expect, these changes to the array's use occur infrequently, at changes of program phase, although the pointers of different types are all created at initialization. This violates liballocs's view of the storage's type—but is arguably a reasonable practice and is done in a disciplined, documented way within the code, so is counted an "unhelpful" false positive. Warnings in such cases may be avoided fairly easily, either by using a union (as with lbm above) or by exploiting the facility of liballocs to dynamically change a chunk's type. This change can be signalled to liballocs with a realloc() call, sized using the new type but totalling the same number of bytes (or fewer). We did not apply any fix in bzip2, since doing so would require refactoring the code somewhat, and the number of false warnings is small.

gcc In this codebase, compiling via CIL required us to refactor one K&R-style function into modern C. gcc's use of alloca() stressed our stack handling, and its use of a garbage-collected heap (nested under malloc()) required the slower generic allocator index (§7.2). Therefore gcc is something of a worst case for performance. It is also the largest codebase, including at least six different internal allocation APIs. Among the libcrunch warnings, several identified tiny bugs involving bad sizeof (like in h264ref) and transposed xcalloc() args (which defeated our allocation site classifier, as predicted in §4) and were fixed. The bad sizeof had the side effect of incorrectly typing a generically-allocated array of pointers in a hash table implementation, causing secondary errors (at four unique locations) as writes into the hash table, made through GPPs (§6.5), were flagged as type-incompatible; these disappeared after the sizeof bug was corrected. Currently our classifier still fails on a small fraction of allocation sites, accounting for about 15% of the heap checks attempted by libcrunch. The remaining failing checks are manageably few for such a large codebase, and break down as follows. Firstly, there was one case of "pointer stuffing", i.e. creating a type-incorrect pointer knowing it will not be dereferenced. In this case it stuffs a pointer to a character string (yielded by the XSTR() macro) into the base field intended to point at an rtx structure (intermediate-representation graph node)

```
if (value->kind > RTX_DOUBLE && value->un.addr.base != 0)
  switch (GET_CODE (value->un.addr.base))
    {
    case SYMBOL_REF:
      /* Use the string's address, not the SYMBOL_REF's address,
         for the sake of addresses of library routines. */
      value->un.addr.base = (rtx) XSTR (value->un.addr.base, 0);
      break;
    /* ... */
    }
```

Since this is a highly error-prone coding practice, we consider it helpful to flag.

Secondly, there were a handful of cases of casts to argumentless function pointer types (§6.6); as before, we consider these outdated style and therefore helpful to flag. Thirdly, we saw one peculiar converse case whereby a function accepting no arguments was passed (following a cast) into a context where arguments *are* passed at the indirect call. This passed function immediately aborts the program if called (in our SPEC runs it is *not* called), so the type-incompatibility is usually harmless. However, this code is not compliant modern C, so the warning is counted as a "helpful" false positive. Fourthly and finally, there was the case of "short-allocated union".

```
union tree_node
{
  struct tree_common common;
  /* ... */
  struct tree_list list;
  /* ... */
 };
typedef union tree_node *tree;
tree tree_cons (purpose, value, chain)
      tree purpose, value, chain;
{
  tree node;
  node = ggc_alloc_tree (sizeof (struct tree_list));
  /* ... */
}
```

Here the code allocates only enough space for a struct tree_list, which is one particular member of union tree_node and is much smaller than some of the others. It then immediately casts it to the bigger type. This saves memory, and is permitted by C's "effective type" rules, but is ob-

benchmark	compile fixes	run-time false positives			
		instances	unique (of which…)		
			total	unhelpful	fixed
bzip2	0	48	3	3	3
gcc	1	3×10^5	14	3	11
gobmk	0	0	0	0	0
h264ref	2	27	2	0	0
hmmer	0	0	0	0	0
lbm	0	5×10^7	8	0	0
mcf	0	0	0	0	0
milc	0	0	0	0	0
sjeng	0	0	0	0	0
sphinx3	0	0	0	0	0

Table 1. Summary of false positives. "Helpful" counts warnings that reflect real defects in the code, albeit not behavioural bugs; examples include calloc() argument transposition or wrong-type sizeof errors. "Unhelpful" includes marginal or unhelpful warnings (e.g. on disciplined re-use of heap memory, as in bzip2). The table shows both dynamic counts (instances) of failed checks, and unique check-failing points in the code, both summed across all of the SPEC workload's invocations of the program. "Unfixed" counts those left standing in the code (cf. the fixes shown in Appendix A) before performance measurement (Table 2).

viously unsafe and error-prone, so we consider it helpful to flag.[9]

Other codebases besides SPEC We have successfully built and run many other codebases under libcrunch, including tcc and git (used in examples earlier in the paper), ghostscript, libpng, lcms, x11-apps and others. Our experiences with these have not uncovered any issues divergent from those seen with SPEC. Common obstacles to using libcrunch are unworkroundable CIL bugs (encountered with eglibc and ffmpeg) or small amounts of C++ code in mostly-C codebases (e.g. ncurses). We already have early-stage Clang and (source-to-source) C++ front-ends which promise to avoid these problems.

8.2 Performance Results

The run-time overheads of our techniques are dependent on many details, not least the host architecture (e.g. regarding TLB overhead of our virtual memory techniques), and our implementation could certainly be optimised further. Rather than definitively quantifying its overhead, our goal in measuring overhead is to justify the claim that the overhead is low enough for a developer to leave it "on by default" for the code they are developing. Application code is instrumented; the host C library is not. Data were intentionally collected on a developer-class machine (Lenovo Thinkpad T420s, Intel i7-2640M quad-core, 4GB memory) running a recent Ubuntu GNU/Linux operating system using gcc 4.9.2. Similarly, we deliberately use SPEC's smaller test workload sizes, to model the intended development-time "edit-run-debug" use pattern; the startup overhead of liballocs is small but measurable, so it would be over-generous to amortise it on unrealistically long jobs. Table 2 shows our execution time measurements. Figures are the median of five runs, taken with all inessential background tasks stopped. Inter-run variation was extremely low—most benchmarks varied no more than 2% from the median, except milc and bzip2 (up to 5%). Note carefully that the crunch slowdown is *only* observed when libcrunch is loaded; for a libcrunch-built binary run normally, the (much lower) slowdown is shown in the nopreload column. (The gcc nopreload case currently crashes, owing to a bug in glibc's dynamic linking of weak thread-local symbols.)

[9] A future extension of libcrunch checking union access would be useful here, but would need to implement an additional, non-obvious check: on any write to a union member, a check that the underlying storage was large enough to accommodate it.

bench	normal/s	crunch/s	crunch %	nopreload	onlymeta
bzip2	4.95	5.29	+6.8%	+1.4%	+2.6%
gcc	0.983	2.58	+160 %	– %	+14.9%
gobmk	14.6	16.1	+11 %	+2.0%	+4.1%
h264ref	10.1	10.5	+3.9%	+2.9%	+0.9%
hmmer	2.16	2.34	+8.3%	+3.7%	+3.7%
lbm	3.42	3.75	+9.6%	+1.7%	+2.0%
mcf	2.48	2.77	+12 %	(−0.5%)	+3.6%
milc	8.78	12.1	+38 %	+5.4%	+0.5%
sjeng	3.33	3.38	+1.5%	(−1.3%)	+2.4%
sphinx3	1.60	2.02	+13 %	+0.0%	+8.7%

Table 2. Run-time performance results: "normal" execution time in seconds, the same under libcrunch, and as a slowdown percentage. The last two columns offer comparisons: "nopreload" is the slowdown of a libcrunch-built binary when libcrunch is *not* loaded, and "onlymeta" is the slowdown of loading libcrunch without having built with it, i.e. of maintaining, but not using, allocation metadata (for static, stack and malloc() heap storage; excludes nested allocators).

Unsurprisingly, the incurred slowdown depends hugely on the density of checks during execution. The sjeng case, which runs marginally faster when compiled with crunchcc, executes only four checks, while gcc performs over 8 million casts in less than a second. Interestingly though, lbm has an even greater check density (49 million in about 3 seconds) but runs with little overhead. This difference owes largely to cacheability: gcc is a heavy user of the stack, including alloca(), but libcrunch's inline check path cannot safely cache stack metadata (§7.1). By contrast, lbm dispatches nearly all its checks on a single heap allocation, which stays at the top of the cache, greatly speeding up checks.

9. Related Work and Discussion

Table 3 summarises some existing systems intended to catch some of the errors we listed in §2; only systems relying partly or wholly on dynamic analysis are included. Note that no tool (besides libcrunch) dynamically checks pointer casts. As we discussed earlier, existing tools such as Soft-

		checking				implementation			compatibility		
system	citation	static?	spatial?	temporal?	type?	metadata gran.	check on	slower	source	binary	catches Fig. 1?
Cyclone	Jim et al. [12]	part	yes	yes	yes	some pointers	ptr create, deref	−10–200%	no	no	yes
CCured	Necula et al. [21]	part	yes	use GC	physical	some pointers	ptr create, deref	0–150%	no	no	depends
Safe-C	Austin et al. [1]	part	yes	yes	no	all pointers	ptr deref	2–7×	yes	no	no
SoftBound[1]	Nagarakatte et al. [18]	no	yes	no	no	pointer	ptr deref	10–350%	yes	yes	no
CETS	Nagarakatte et al. [19]	no	no	yes	no	allocation, pointer	ptr deref	0–170%	yes	yes	no
Memcheck	Seward and Nethercote [24]	no	coarse	yes	no	bit	value create	10–55×	yes	yes	no
ASan	Serebryany et al. [23]	no	coarse	partial	no	pointer	ptr deref	10–170%	yes	yes	no
Hobbes	Burrows et al. [3]	no	coarse	no	physical	word	value create	90–190×	yes	yes	maybe, in callee
unnamed	Loginov et al. [16]	no	coarse	no	physical	byte	value use	5–135×	yes	yes	maybe, in callee
SAFECode	Dhurjati et al. [6]	part	partial	partial	partial	allocation pool	ptr create	0–30%	yes	no	depends
libcrunch	this paper	no	no	no	yes	allocation	ptr create	0–160%	yes	yes	yes

[1] for consistency, fields refer to SoftBound's "full checking" mode

Table 3. Some existing dynamic (or part-dynamic) analyses for catching type- and/or memory-related errors in unsafe code.

Bound [18] and Memcheck [24] are focused on memory properties which we do not check, while heavyweight dynamic type checkers [3, 16] work by tagging words in memory, and do not model user-defined data types. We can also distinguish existing systems by what granularity of metadata they maintain (per word, per allocation, etc.), and what classes of primitive operation they interpose on (pointer dereference, pointer cast, etc.). The low overhead of libcrunch is explained by the relative infrequency of the operations it intercepts, and (relatedly) the relatively coarse-grained metadata that suffices. As we discuss next, other work takes hugely varying approaches to source- and binary-level compatibility: perhaps restricting or outright modifying the the source language (no compatibility), perhaps keeping source unchanged but forgoing binary compatibility (as with traditional "fat pointer" approaches) through to preserving the binary compatibility also (as with disjoint metadata approach like libcrunch's or SoftBound's).

Hybrid static/dynamic approaches Modified source languages can offer partially static checking, hence low runtime overheads, but at the expense of imposing on the programmer. CCured [21] is a dialect of C which enforces run-time type- and memory-safety properties, performing much of its reasoning statically but with a dynamic fallback. It sacrifices binary compatibility in most cases (when "wild" pointers are needed), requires wrapper annotations for calls to external libraries, and imposes a conservative garbage collector. Cyclone [12] eschews the latter in favour of region-based memory management, but also diverges more markedly from C.

Physical typing CCured adopts "physical typing" as a basis for type checks [4, 25]. This means treating memory as a list of primitive values (pointers, integers, floating-point numbers), which can discard arbitrarily deep structures. Consider a deep nest of structs, where at the leaves all fields are double: physically this is merely an array of double, but this view removes a lot of information. Although this naturally accommodates "loose" C idioms, like casting struct pointers between unrelated target types, it is extremely imprecise and can discard meaningful data abstraction.

Dynamic complement to alias analysis SAFECode [6] combines whole-program static analysis with just enough dynamic checking to enforce properties inferred by static analysis: that memory errors at run time do not invalidate the points-to class of a given pointer. Points-to classes can encode object types. However, the checking is very partial: a hard-to-analyse codebase, for which static analysis is inconclusive, leads to *fewer* dynamic checks, hence more uncaught errors.

Heavyweight dynamic approaches Hobbes [3] and the system of Loginov et al. [16] use heavyweight instrumentation to attach physical types to machine words. Unfortunately, this forbids the legal conversion of pointers to integers and back, as noted by Yong and Horwitz [27], and the slowdown is considerable (of the order 10x–100x).

Type qualifiers We discard C's const and volatile. While we could perhaps exploit them, the issue is subtle. A const qualifier on a pointer type like C's const char * exists to restrict the *user* of the object (it may not modify the target chars) rather than the range of objects pointed to (it may point to objects that were or were not *allocated* as const). By contrast, volatile generally does refer to allocations. The applicability of other qualifiers explored in various research work [5, 9] is likely to vary similarly, and is worth investigating. Dynamic checking could prove useful, since enforcing static checking of qualifiers like const means onerous "transitive" changes when they are introduced to a codebase (referred to as "const poisoning" at the time of const's introduction).

Debug-time analysis Polishchuk et al. [22] present an analysis of memory image snapshots which attempts to infer a typing for every allocated region of the memory image. Like our work, it exploits debugging infrastructure and instruments the program's allocators. Unlike ours, the analysis is invoked from a debugger, rather than running continuously during execution. Another key difference is that it makes certain assumptions that we avoid: that "no non-pointer ever takes on a valid pointer value by chance" (prone to failure in large programs), that a heap block's size is an exact multiple of its element size (falsified many cases of padded or

variable-length objects we have seen in practice) and has no nested allocators. Plumbing libcrunch into a debugger like gdb would be very worthwhile.

Executable C semantics As noted, earlier executable C semantics such as CH_2O [14], Cerberus [17] or kcc [8] are valuable but essentially complementary efforts. They are exhaustive, highly detailed tools likely to prove useful for establishing strong correctness and/or portability properties of small pieces of C code. By contrast, our system is a pragmatic tool that developers can reasonably leave enabled during everyday development on large codebases. We are currently working on stating the properties libcrunch checks in terms of the semantic framework of Cerberus.

Approaches to heap metadata Our runtime relies on carefully crafted *index* structures to look up allocations' type metadata. Another family of approaches to associating metadata with heap allocations is based on *placement*, i.e. the allocator encodes metadata by its choice of object address, as in the "big bag of pages" allocator (described by Wilson et al. [26]). Our approach has some benefits regarding compositionality: whereas placement inherently supports only one placement policy at a time, multiple heap-threaded indexes can index the same set of chunks in different ways. It can also store more metadata without exhausting the available address space. By contrast, placement can inherently encode only a fraction of a word (i.e. the set of address ranges available for issue) and quickly absorbs large regions of the VAS.

10. Conclusions

We have presented libcrunch, a system which complements previously existing dynamic checking in unsafe code by checking type correctness properties with high precision and low run-time overhead. The obvious next question to ask is: can we use these techniques, in combination with known spatial and temporal memory checks, to create a memory-*and* type-safe variant of C that is source-compatible with a large fraction of existing code? We believe so. Even without this, as we have shown, the techniques in libcrunch enable a valuable tool for working programmers debugging difficult-to-understand failures arising from pointer casts and similar errors.

Acknowledgments

This work was supported by EPSRC Programme Grant "REMS: Rigorous Engineering for Mainstream Systems", EP/K008528/1. I am grateful to Stephen Dolan and Leo White for insights which greatly helped in figuring out the right treatment of polymorphic code. I am also grateful to Peter Sewell for comments on a draft, and to the anonymous reviewers for their many helpful suggestions.

References

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 290–301, New York, NY, USA, 1994. ACM.

[2] G. Banavar, G. Lindstrom, and D. Orr. Type-safe composition of object modules. Technical Report UUCS-94-001, University of Utah, Salt Lake City, Utah, USA, 1994.

[3] M. Burrows, S. N. Freund, and J. L. Wiener. Run-time type checking for binary programs. In *CC'03: Proceedings of the 12th international conference on Compiler construction*, pages 90–105, Berlin, Heidelberg, 2003. Springer-Verlag.

[4] S. Chandra and T. Reps. Physical type checking for C. In *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '99, pages 66–75, New York, NY, USA, 1999. ACM.

[5] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 85–95, New York, NY, USA, 2005. ACM.

[6] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 144–157, New York, NY, USA, 2006. ACM.

[7] C. Diamand, S. Kell, and D. Chisnall. Run-time type checking with clang, using libcrunch. Presented at EuroLLVM 2016, March 2016. Presentation abstract, slides and video available at http://www.llvm.org/devmtg/2016-03/ as retrieved on 2016/8/26.

[8] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 533–544, New York, NY, USA, 2012. ACM.

[9] J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, Nov. 2006.

[10] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter 1992 USENIX Conference*, pages 125–138. USENIX Association, 1991.

[11] ISO WG21. Programming languages — C. ISO/IEC Standard 9899:2011, Dec. 2011. A non-final but recent version is available at www.open-std.org/JTC1/SC22/WG14/www/docs/n1539.pdf, retrieved on 2016/8/24.

[12] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[13] S. Kell. Towards a dynamic object model within Unix processes. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Soft-*

ware (Onward!), Onward! 2015, pages 224–239, New York, NY, USA, 2015. ACM.

[14] R. Krebbers and F. Wiedijk. A typed C11 semantics for interactive theorem proving. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, pages 15–27, New York, NY, USA, 2015. ACM.

[15] S. Krishnamurthi and M. Felleisen. Safety in programming languages. Technical Report TR 99-352, Rice University, 1999.

[16] A. Loginov, S. H. Yong, S. Horwitz, and T. W. Reps. Debugging via run-time type checking. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, FASE '01, pages 217–232, London, UK, UK, 2001. Springer-Verlag.

[17] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell. Into the depths of c: Elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 1–15, New York, NY, USA, 2016. ACM.

[18] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[19] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. In *Proceedings of the 9th International Symposium on Memory Management*, ISMM '10, pages 31–40, 2010.

[20] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. N. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer Berlin Heidelberg, 2002.

[21] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 128–139, New York, NY, USA, 2002. ACM.

[22] M. Polishchuk, B. Liblit, and C. W. Schulze. Dynamic heap type inference for program understanding and debugging. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 39–46, New York, NY, USA, 2007. ACM.

[23] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[24] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *ATEC '05: Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2005. USENIX Association.

[25] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, pages 180–198, London, UK, UK, 1999. Springer-Verlag.

[26] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proc. International Workshop on Memory management*, pages 1–116. Springer Verlag, September 1995.

[27] S. H. Yong and S. Horwitz. Reducing the overhead of dynamic analysis. *Electr. Notes Theor. Comput. Sci.*, 70(4):158–178, 2002.

[28] Q. Zhao, D. Bruening, and S. Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 22–31, New York, NY, USA, 2010. ACM.

A. Code Changes to SPEC CPU2006

This Appendix briefly lists all the changes to the SPEC CPU2006 code that were made before measuring performance results. They incorporate compile fixes, and, following libcrunch's warnings, minor changes to fix and/or silence false-positive warnings (both helpful and unhelpful) in the original code. A full patch is available in the accompanying artifact.

gcc

- c-common.c line 2934: rewrite K&R prototype as modern C.
- hashtab.c lines 311 and 318: use intended type in sizeof (PTR not PTR *).
- reload1.c line 3504: fix transposed xcalloc() arguments.
- sbitmap.c line 63: use intended type in sizeof (sbitmap not sbitmap *).
- tree.c line 787: reorder summation of sizeof applications to reflect in-memory order.

h264ref

- image.c line 65 and 1706: fill in argument types in function declaration and definition.
- mv-search.c line 1092: replace known-buggy (undefined) code with fix from gcc bugzilla (http://gcc.gnu.org/bugzilla/show_bug.cgi?id=53073).
- parsetcommon.c lines 35, 37, and 56: fix transposed calloc() arguments.
- memalloc.c lines 304 and 552: use intended type in sizeof (short *** not short **).

lbm

- lbm.c and lbm_1d_array.h: introduce union double_or_uint as detailed in §8.1.

B. The Problems of perlbench

We noted in Section 8 that perlbench, a version of the Perl scripting language interpreter, was the only in-scope SPEC CPU2006 benchmark that our tool could not usefully be applied to. Here we discuss what currently goes wrong, and how it can likely be fixed.

B.1 Loose Structural Aliasing

The core of Perl is built on a family of struct types with similar (but not identical) field prefixes. The following real code excerpt illustrates.

```
/* Using C's structural equivalence to help emulate C++
 * inheritance here... */

struct sv {
    void* sv_any;   /* pointer to something */
    U32   sv_refcnt; /* how many references to us */
    U32   sv_flags;  /* what we are */
};
struct gv {
    XPVGV* sv_any; /* pointer to something */
    U32   sv_refcnt; /* how many references to us */
    U32   sv_flags;  /* what we are */
};
struct cv {
    XPVCV* sv_any; /* pointer to something */
    U32   sv_refcnt; /* how many references to us */
    U32   sv_flags;  /* what we are */
};

/* snip */

struct xrv {
    SV *   xrv_rv;    /* pointer to another SV */
};

struct xpv {
    char * xpv_pv;   /* pointer to malloced string */
    STRLEN xpv_cur; /* length of xpv_pv as a C string */
    STRLEN xpv_len; /* allocated size */
};

struct xpviv {
    char * xpv_pv;   /* pointer to malloced string */
    STRLEN xpv_cur; /* length of xpv_pv as a C string */
    STRLEN xpv_len; /* allocated size */
    IV      xiv_iv;     /* integer value or pv offset */
};
```

We cannot use the usual treatment of structure prefixing, based on __like_a() (§6.1), since, for example, a gv is unfortunately not __like_a sv. This is because its first field has an incompatible type (not void*): whereas the first field in sv can hold any pointer, in gv it must point to an XPVGV. So we cannot cast the address of a gv to sv* without risking unchecked writes that violate gv's contract. But note that our treatment of parametric-style polymorphism using void** solved (§6.5) almost the same problem: storage holding a T* cannot be treated __like_a void* because clients of the latter won't enforce the former's contract. We fixed this by allowing void** to point more widely (to any safely-readable value) while dynamically enforcing the actual contract of the pointed-to storage (by instrumenting writes). The same approach generalises to sv*: a pointer-to-sv that actually points at a gv would be read-safe (since reading the pointer field as void* is harmless) but would require additional checks on writes (to enforce the underlying gv's stronger contract on its pointer field).

We therefore re-formulate our "pointers to generic pointers" treatment in terms of pointers to *generic-pointer-containing object types* (GPCOTs). An object of type void* is a generic-pointer-containing object, but so is a struct having a void* element. In general, a generic-pointer-containing object type is defined recursively as

- a generic pointer type; or
- (recursively) a struct or union type having a field that is a GPCOT; or
- (recursively) an array type whose element type is a GPCOT.

Casts whose target type is a pointer to some GPCOT type T are checked using the __loosely_like_a(_, T) predicate. This is like __like_a but, for any subobject in T that is a generic pointer, matches *any pointer* subobject in the actual object, so long as it has at least the degree of the generic pointer. This allows a T* to be viewed as a void*, or a S*** to be viewed as a void**, while the degree check prevents, say, an int* from being viewed as a void** and then double-dereferencing (which would dereference an int).

As with GPPs, and again in a departure from our usual check-on-create, this relaxation requires all *writes of pointers through an lvalue of GPCOT type* be instrumented, such that the written value is contract-checked against the underlying object's stored type. The recursive definition ensures that it is not possible to bypass this write-checking by taking the address of (any part of) an lvalue of GPCOT type; that pointer, too, will be (by construction) a pointer-to-GPCOT so any writes through it must be checked.

We have implemented this refinement in libcrunch's instrumentation, and it does indeed eliminate a large fraction of perl's false positives—but far from all, as the next section notes.

B.2 Fudged Allocations

These structure types are allocated out of many arenas that are obtained from malloc(). Each arena holds only instances of a single struct type. For example, the following shows the code for allocating an xpv-holding arena. The while loop implements an initialization convention for unused structures: the first pointer in a free structure points to the next such structure in the arena, i.e. the arena is initialized to a free list.

```
/* allocate another arena's worth of struct xpv */
STATIC void
S_more_xpv(pTHX)
{
```

```
    register XPV* xpv;
    register XPV* xpvend;
    New(713, xpv, PERL_ARENA_SIZE/sizeof(XPV), XPV);
    xpv->xpv_pv = (char*)PL_xpv_arenaroot;
    PL_xpv_arenaroot = xpv;

    xpvend = &xpv[PERL_ARENA_SIZE / sizeof(XPV) - 1];
    PL_xpv_root = ++xpv;
    while (xpv < xpvend) {
        xpv->xpv_pv = (char*)(xpv + 1);
        xpv++;
    }
    xpv->xpv_pv = 0;
}
```

This is something libcrunch is very capable of dealing with—the New macro expands to a malloc() invocation with appropriate sizeof, so we can infer that the object holds XPV instances. Unfortunately, almost all the *other* structs' such arenas are allocated in a perverse way: by first creating an arena to hold XPV, and then "fudging" it to in fact initialize the pointers using the alternative struct size.

```
STATIC void
S_more_xiv(pTHX)
{
    register IV* xiv;
    register IV* xivend;
    XPV* ptr;
    New(705, ptr, PERL_ARENA_SIZE/sizeof(XPV), XPV);
    ptr->xpv_pv = (char*)PL_xiv_arenaroot; /* linked list of xiv arenas
          */
    PL_xiv_arenaroot = ptr;                    /* to keep Purify happy */

    xiv = (IV*) ptr;
    xivend = &xiv[PERL_ARENA_SIZE / sizeof(IV) - 1];
    xiv += (sizeof(XPV) - 1) / sizeof(IV) + 1; /* fudge by size of XPV */
    PL_xiv_root = xiv;
    while (xiv < xivend) {
        *(IV**)xiv = (IV *)(xiv + 1);
        xiv++;
    }
    *(IV**)xiv = 0;
}
```

As expected, this causes lots of casts to XIV (and other types) to fail, thinking that they are pointing at allocations of xpv. The only solution is to rewrite the code to avoid fudging.

B.3 Type Descriptors

A final challenge in supporting Perl is the fact that some allocations are not sized near the allocation site, but instead using a size value that was *stored* for later pick-up. The Newc macro invocation in the following code retrieves a size from a PerlIO_funcs object (in the expression tab->size).

```
if (tab->size) {
    PerlIOl *l = NULL;
    if (tab->size < sizeof(PerlIOl)) {
        goto mismatch;
    }
    /* Real layer with a data area */
    Newc('L',l,tab->size,char,PerlIOl);
    // ...
```

What are these sizes? A host of PerlIO_funcs instances are defined statically, of the following form.

```
PerlIO_funcs PerlIO_remove = {
    sizeof(PerlIO_funcs),
    "pop",
    0,
    PERLIO_K_DUMMY | PERLIO_K_UTF8,
    PerlIOPop_pushed,
    NULL,
    NULL,
    NULL,
    // ...
    NULL,                       /* flush */
    NULL,                       /*  fill */
    NULL,
    // ...
};
```

As it happens, the stored size is *always* sizeof(PerlIO_funcs), but there is no easy way for us to infer this. One way to fit this into liballocs's allocation-site-centric approach is to view the passed-in pointer to a PerlIO_funcs as a *size descriptor*, where each descriptor has an identity and link time and maps to a specific size. These can then be enumerated by the user, in an environment variable, using their linkage name, much like the user already enumerates allocation functions. This is a little tedious, however. It would be possible to propagate the dimensioned sizeof quantities as far as link time, hence recording the fact that the first field of each PerlIO_funcs is initialized to some value having a sizeof dimension. This is then enough to infer the sizeofness associated with the PerlIO_remove object, say, and can be seen as a link-scope generalisation of the (function-local) static analysis we currently perform (§4). The most precise approach would pass dimensioned size quantities around at run time; when they could perhaps be represented using pointers to our uniqtype objects. However, this propagation would have significant cost.

B.4 Prospects of Perl running under libcrunch

With enough effort to accommodate the above difficulties, and fixing of the egregious "fudging" code, we believe it is possible to have Perl running smoothly under libcrunch—although that remains to be seen. The hardest part to dealing with all the above is to understand it—to unpick the arcane allocation and subtyping schemes that Perl's authors have created. Consequently, one expects the effort required by developers to use libcrunch routinely in their work to be much lower than that required when a third party (here the present author!) does so from a cold start. Even if Perl were not an extreme case in the extent of its allocation and type-punning quirks, the effort involved for its own developers, who already understand these arcana, would be significantly reduced.