Fast Run-time Type Checking of Unsafe Code

Stephen Kell

Computer Laboratory, University of Cambridge
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
firstname.lastname@cl.cam.ac.uk

Abstract

Existing approaches for detecting type errors in unsafe languages work by changing the toolchain's source- and/or binary-level contracts, or by imposing up-front proof obligations. While these techniques permit some degree of compile-time checking, they hinder use of libraries and are not amenable to gradual adoption. This paper describes libcrunch, a system for binary-compatible run-time type checking of unmodified unsafe code using a simple yet flexible language-independent design. Using a series of experiments and case studies, we show our prototype implementation to have acceptably low run-time overhead , and to be easily applicable to real applications written in C without source-level modification.

1. Introduction

C, C++ and other unsafe languages remain widely used. "Unsafe" means that the language does not enforce memory- or type-correctness invariants, such as "all pointer use will respect the bounds of the object from whose address the pointer derives" (spatial memory-correctness), "an object is only reclaimed once no live pointers into it remain in circulation" (temporal memory-correctness) or "reads (and writes) to an object will select interpretations (resp. representations) consistent with the object's associated data type" (type-correctness).[1]

[1] Readers will note that this is a narrower notion than the concept of "type" as originating in symbolic logic and often applied to programming languages. We make no reference to this notion; rather, we are concerned only with "data types", and our definition is somewhat modelled on that of Saraswat [1997]. We elaborate on these distinctions in §8.1.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]. . . $10.00

In return, unsafe languages offer several benefits: avoidance of certain run-time overheads; a wide range of possible optimisations; enough control of binary interfaces for efficient communication with the operating system or remote processes. These are valid trades, but have so far come at a high price: of having *no* machine-assisted checking of type- or memory-correctness beyond the limited efforts of the compiler. Various tools have been developed to offer stronger checks, but have invariably done so by abandoning one or more of unsafe languages' strengths. Existing dynamic analyses [Burrows et al. 2003; **?**] offer high run-time overhead, while tools based on conservative static analyses [**?**] are unduly restrictive. Hybrid static/dynamic systems offering low-to-moderate run-time overhead [**??**] invariably sacrifice compatibility with existing code at either source level (entailing porting effort) or binary level (precluding use of pre-built libraries). The most practically compelling tools [Nagarakatte et al. 2009; Seward and Nethercote 2005; **?**] have focused on memory correctness and do not offer any checks over data types.

This paper describes a run-time system which enables *dynamic type checking* within unsafe code, while maintaining full binary compatibility (for easy use of libraries), requiring no source-level changes (as we demonstrate in the C front-end to our system), and offering sufficiently low overheads that it can comfortably be left enabled "by default" during development. Our research contributions are:

- a language-independent model of *allocations* and *data types* which captures the essential aspects of program behaviour while integrating conveniently into existing toolchains;
- the design of a C front-end, dealing pragmatically with C's untyped heap allocation, imprecisions in common programming styles, and other subtleties;
- an efficient implementation based around novel disjoint metadata techniques;
- benchmark results and case-study experiences substantiating our claims that our system offers good performance

```
if (obj↪type == OBJ_COMMIT) {
  if (process_commit(walker, (struct commit *)obj))
    return -1;
  return 0;
}
```

Figure 1. A typical pointer cast in C code (from the git version control system). The programmer believes that obj's target can be interpreted as a commit, but no check is done.

```
if (obj↪type == OBJ_COMMIT) {
  if (process_commit(walker,
        assert(__is_a(obj, "struct␣commit")),
          (struct commit *)obj))
    return -1;
  return 0;
}
```

Figure 2. A sketch (for exposition purposes; not illustrative of our implementation) of how the git example pointer cast is treated by libcrunch and crunchcc

properties, is easy to apply to real codebases, and yields useful feedback to programmers.

2. Approach

Our approach is a pragmatic one, informed by several observations about practical software development, as we now outline.

Checks as assertions Similar to the perspective of **?**, we interpret source code as containing latent specifications; our tool simply lifts these into something explicit which can be checked. Fig. 2 shows a real code fragment (from git[2] which could benefit from our tool, and Fig. 1 shows an expositional view of how it is treated by our tool.

Toolchain extension An essential philosophy of unsafe languages is that the core language is deliberately liberal, and the programmer is responsible for finding (or building) whatever restrictions, including coding practices and tool-imposed policies, that are appropriate to the task at hand. Such programmers are accustomed to deploying a selection of tools over and above what is provided by a basic compiler and runtime. Many such tools for debugging memory errors are widely used, including Purify [**?**], Memcheck [Seward and Nethercote 2005], mudflap[3] or other Jones & Kelly-style checkers [**?**], Address Sanitizer [**?**] and so on. Our tool offers a similar user experience to these, while checking type-based properties, and with generally much lower run-time overheads.

Access to source code Unlike certain memory properties, type-based properties benefit from access to source code, since useful source-level information is compiled away early. We will exploit this in our C front-end (§5) where the programmer's use of the sizeof operator is a key input.

No modifications to source code Practical adoption of a tool is greatly eased when its use does not require the programmer to annotate or modify their source code. FIXME: can reference something about this? Although our implementation will make use of some source-to-source translation, this is fully automatic and used only to the extent that it is necessary; we also employ various link-time and runtime techniques to insert our checking into the program; we prefer since they are less invasive and less language-specific.

Partial programs and binary compatibility Practical adoption of a tool is also eased by avoiding any requirement to recompile the whole program—particularly libraries. A corollary of this requirement is that binary interfaces of code compiled with and without the tool should not differ. This allows any combination of "with"- and "without"-compiled code to be linked together as normal. We consider the consequences of various combinations of instrumented and uninstrumented code (§4.3).

Distinguish type- from memory-correctness We focus our tool solely on the problem of checking type correctness. This owes partly to the fact that whereas various memory checkers are widely used, we know of no particularly usable tool for checking type correctness. Meanwhile, since memory-incorrectness risks arbitrary state corruption, we in effect check type-correct execution *assuming* execution has so far been memory-correct, expecting that the programmer will use one of the previously cited memory checkers in combination with our tool. This separation allows us to produce a much lower-overhead tool than a precise (subobject-sensitive) memory checker, since the latter is forced to track *per-pointer* metadata, whereas we only require *per-allocation* metadata. We can also opportunistically catch some memory errors, since memory-incorrect code is often type-incorrect. (We will sketch, albeit only as future work, how one could combine both kinds of checking in a way that shares work where possible.) FIXME: mention any data on opportunism if/when we have any.

Low overhead, "on by default" Our tool has low enough overhead (at most 20% of execution time except in rare cases, often lower, and with similarly low memory overheads) that it can be enabled by the developer as a matter of routine without any incurring noticeable slowdowns in workflow.

Bug-finding over hard guarantees Since much of the program may have been compiled out of sight of our tool, we cannot rule out misbehaviour of that code, and so cannot

[2] FIXME:git

[3] http://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging

guarantee whole-program invariants. Put differently, our focus on ease of adoption, leading us to ref has the trade-off that we must relax any attempt to provide guarantees; our tool is primarily a bug-finding aid rather than a run-time *verification* of the strength desired for safety-critical or high-security applications. (Nevertheless we will consider, informally, what guarantees are available the special case where *all* code in a process has been instrumented in a particular fashion—§5.1.)

De-emphasise language semantics On the (somewhat philosophical) issue of "when should a check fail?", we take our lead from what large population of programs do, rather than by slavishly adhering to any one language specification (say, the C language). For example, the C standard's notion of "effective type" is approximately the same notion as the one we attach to memory, but there are some important differences (§5.6) which reflect real (albeit technically incorrect) C coding practices.

Mechanism over policy We seek to provide general, efficient mechanisms while leaving policy somewhat user-customisable. This is reflected in how our system is factored into a run-time library, libcrunch, providing the mechanisms—which we consider the primary value of the system—whereas per-language front-ends, such as crunchcc, which are concerned more with policy, and are kept simple and separate, with the intention of being customisable.

Language-independence Our design has an ulterior motive: to form part of an infrastructure which can provide a range of run-time services, not just type checks, and which can underlie not only unsafe code, but *all code* in the user-level software stack, of which the unsafe case is general (and hard) case. We therefore pursue a language-independent design for libcrunch, where any language-specific code is minimised and isolated. We intend future iterations of our infarstructure to be useful to implementors of other languages, as a foundation for efficient multi-language runtime environments (whose languages would include, but not be limited to, unsafe languages such as C and C++) and multi-language tools (including debuggers, profilers and the like). This contrasts with the status quo, in which cross-language development is painful for a variety of reasons, and where even on supposedly multi-language platforms such as the CLR, JVM and so on, C and other unsafe languages receive limited support—despite the huge investment constituted by the collection of code written in these languages.

Compile-time assistance We assume that the compiler prevents type errors in data movements contained within a single function activation's actual parameters, local variables and temporaries. For example, given a local which might be declared in C as int and another as float, we assume that the compiler does not blindly allow a floating point bit-pattern to be copied into the integer—that it will insert a conversion or insist on the user inserting one. The other data movements are precisely those that require storage to be *address-taken*. This assumption allows us to focus entirely on use of pointers (including pointers to functions), since those accesses which may reach storage that the compiler cannot reason locally about are precisely those that necessitate the use of pointer values at run time.

3. Design overview

In this section we summarise the key insights behind our design, and explain how it extends a conventioanl compilation toolchain.

3.1 Insights

Our design is based on three insights, each of which covered in a subsequent section of the paper.

Allocation sites —we observe that when suitably defined, these *necessarily* tell us what data type(s) a given piece of memory may be interpreted as, and can therefore be the basis of a dynamic analysis associating memory with data types. More specifically, we find that a hierarchical model of allocation is necessary (§4.1).

Debugging information —we observe that debugging information offers a ready-made language-independent notion of data types which can conveniently be augmented with allocation site information to describe a whole program's view(s) of its data (§4.2).

Mechanism–policy split —the language-specific parts of such a system are surprisingly easy to isolate. A language-independent run-time and toolchain extensions provide the important mechanisms supporting a variety of policy decisions, mainly concerning *what* to check and *when* to do so, can be taken on a per-language basis. Our C front-end explores a variety underneath "C" exists considerable variation in programming style—strict versus sloppy use of abstractions, procedural- versus indirection-heavy "O-O"-style code, etc. (§5).

Disjoint metadata schemes —when tailored to different levels in the allocation hierarchy, and combined with similar structures for stack and static storage, these can be used to efficiently index the entire program's address space. Our disjoint metadata schemes have much in common with existing "shadow space" implementations using virtual memory, but build on these techniques by combining a large linear mapping with bucket structures reminiscent of a hash table. These structures allow us to achieve simultaneous efficiency in time, memory and virtual address space usage (§6).

3.2 Toolchain extension

Fig. 3 shows an end-to-end view of how a C program is compiled, linked and executed in our libcrunch-enabled toolchain. The top-level input is unmodified C source code, and the compiler output is binary-compatible with what an

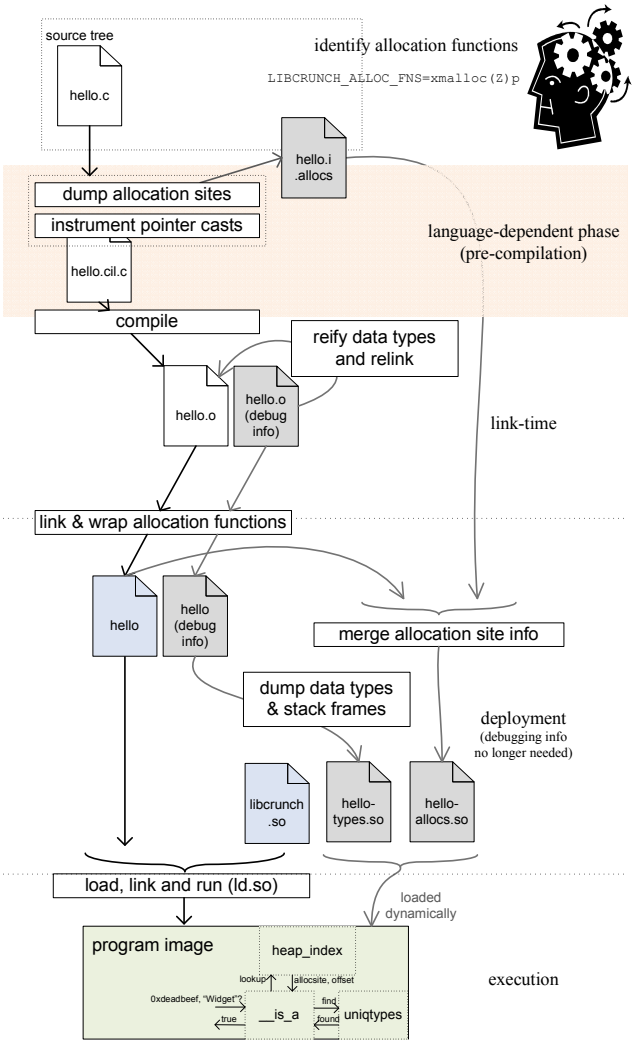

Figure 3. How libcrunch extends the toolchain. Metadata processing is on the right, with grey arrows, and the "normal" compile-link-run path is on the left with black arrows. Only the orange-background part is language-specific. FIXME: highlight the libcrunch extensions in colour somehow. FIXME: use a simplified view of this figure here, then repeat the full one later

ordinary C compiler would output. Metadata processing is on the right, with grey arrows, and the "normal" compile-link-run path is on the left with black arrows. Only the orange-background part is language-specific. Relative to a conventional compiler run, key differences are the gathering of *allocation site information*, based on specifications of which functions are allocation functions. These are the primary guidance which the programmer must supply to libcrunch over and above the source code, but are simple enough that they can be maintained outside the source tree (we tend to store them in Makefiles and/or shrc-style shell mini-scripts).

Builds using our toolchain must always initially generate debugging information, since this is used in intermediate processing stages, but this can be stripped before deployment. The additional deployment-time artifacts, aside from the libcrunch library itself, are *allocation site* and types binaries consisting of program metadata in efficient representations ready for run-time consumption by libcrunch (which loads them in as shared libraries). We discuss these in the following section.

3.3 User's view

Fig. 4 shows what a user of libcrunch might see at their terminal when compiling and running a mildly buggy program under libcrunch. The compiler output can be run without libcrunch, and runs normally with negligible run-time overhead: all checks are diverted into a "no-op" stub libcrunch in which they always succeed. Alternatively, the same binary can be run with libcrunch loaded (here using the LD_PRELOAD option for ELF-based dynamic linkers) which will generate check messages (as the program executes) and summary statistics (at the end).

Several of our design decisions have practical benefits: the system is very debuggable because checks and data types are reified as ordinary code. Programmers can put breakpoints on check functions, watch suspect variables inspect the definitions of data types at run time to understand why an error was provoked. FIXME: any more points to make along these lines? Want to talk about the libcrunch API as a debugging helper?

4. Data types and allocations

Our system necessarily maintains a run-time model mapping pieces of *allocated memory* onto the *data types* of which they store instances. We consider these somewhat intertwined issues in this section, starting with allocations.

4.1 Allocations

Static, stack and heap We divide allocations in the program into the usual three categories: static, stack and heap. Static allocations hold data of whole-program lifetime[4] If we assume a modern Unix-like operating system, static allocations are parts of the program binaries mapped into the address space using *mmap()*, and their contents are described by the binaries' debugging information (which including the locations and signatures of functions, and the locations and data types of variables and constants, down to the details of their field layout and bitwise representation). Stack memory is allocated implicitly by changes to the stack pointer, but in exactly the same way that a debugger uses the stack frame's program counter to understands its layout, we may also walk the stack at run-time and use debugging information to un-

[4] In the presence of dynamic loading and unloading, this lifetime is instead equal to the duration for which the containing object is loaded. This makes no practical difference for our purposes.

```
$ 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
...
libcrunch summary:
checks begun:                                  19203
-----------------------------------------------------
checks aborted for bad typename:                   0
checks aborted for unknown storage:                0
=====================================================
checks remaining                               19203
-----------------------------------------------------
checks handled by static case:                     0
checks handled by stack case:                      0
checks handled by heap case:                   19203
-----------------------------------------------------
   of which did lazy heap type assignment:         0
-----------------------------------------------------
checks aborted for unindexed heap:                 0
checks aborted for unknown heap allocsite:         0
checks aborted for unknown stackframes:            0
checks aborted for unknown static obj:             0
checks failed inside allocation functions:         0
checks failed otherwise:                           1
checks nontrivially passed:                    19202
```

Figure 4. A user's-eye view of libcrunch using the C frontend (crunchcc). Readers who have used Valgrind's Memcheck tool [Seward and Nethercote 2005] are likely to find this user experience familiar.

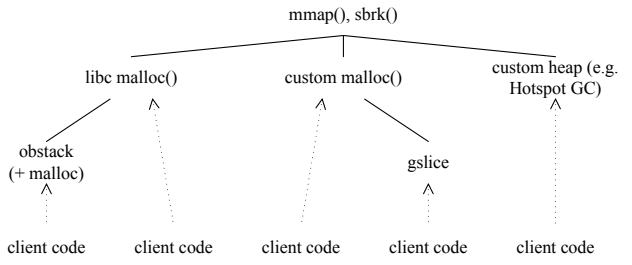

Figure 5. An allocator tree as might be found in a large C/C++/Java program

derstand its contents; we describe an implementation of this in §6.3.

The heap difficulty The remaining case of heap allocations is the most challenging, because of the variety of schemes by which this memory is allocated, by the fact that debugging information tells us nothing about it, and by the fact that the data types instantiated in heap memory are often recorded only implicitly, as with C's malloc() function which takes only a size. Logically speaking, we extend debugging information with information about *allocation sites* which allocate a particular data type. Not all allocation sites allocate a particular data type, but we can combine our knowledge of the ones that do so to track all heap-allocated data type instances in the program, as we now explain.

Allocator trees The collection of heap allocators in a program can be arranged in a tree, where an allocator B is a child of A if B calls A to obtain the memory that it parcels out to its clients. At the top of the tree are the operating system mechanisms—on Unix these are mmap() and sbrk(), which are interchangeable for our purposes. We assume for the moment that each allocator is procedurally abstracted. Fig. 5 shows the different heap allocators that might be present in some reasonably large C and/or C++ program; for variety we also suppose that it contains an instance of a Java Virtual Machine, such as Hotspot. Client code can request allocations at various levels in the tree: from malloc(), or from an allocator layered over malloc() such as the C library's obstack allocator[5] perhaps direct from mmap(), or from a garbage-collecting allocator such as that in Hotspot, and so on. An *allocation event* in a program's execution is a call into an allocator—either from client code or from another allocator. We observe that branch-level allocators size the allocations they request (from their parent) according to performance characteristics (what the typical sizes and lifetimes of allocation are, the cost of an mmap(), the page size, hence expected losses due to fragmentation, etc.). Meanwhile, leaf-level allocations must be sized according to what *data type* they are to hold. The points in execution where memory acquires a data type are the returns from leaf-level allocation calls. (Memory which has been allocated by a branch-level allocation call but not yet by a leaf-level allocation call is not an instance of any data type. Note also that we cannot talk about "leaf-level allocators"—the same allocator might be used directly by client code, such as a client calling malloc(sizeof(foo)), or by a nested allocator; in the former case malloc() is operating at leaf level, and in the latter at branch level. Leafness or branchness is a property of an *allocation site*, not an allocator.)

(One complication is that allocators operating at all levels typically embed their own bookkeeping data structures, which *are* instances of data types, into the memory they request from their parent allocator. For the moment we do not attempt to check type-correctness of the allocator code manipulating these structures, because it is hard to distinguish from operations on untyped memory and would complicate our metadata implementation; allocators are in effect "trusted code". However, we consider some refinements that would allow this in §9.)

Unusual allocation functions This formulation is surprisingly robust, but it can lead to some unexpected code being deemed "allocation functions" in unusual cases. If a function re-uses some memory, it becomes an allocation function. Alternatively, consider a function which allocates using an overapproximate size, in order to hold variable-length piece of data whose length is bounded but not yet known at allocation time. In this case, still, domain knowledge of the data is used to size the allocation; typically, such could would subsequently leave the remaining (over-estimated) portion

[5] Obstacks can use arbitrary higher-level allocator, so more properly we should talk about "obstacks layered over malloc()".

of the allocation unused. If the code instead finds some way to re-use it, then it is itself an allocation function! In practice, whole-object re-use is somewhat common, but the latter kind of opportunistic re-use is highly unusual, because it constrains the lifetimes of the two objects to be identical.

User-supplied hints In summary, leaf-level allocation sites represent the points where a unit of heap storage acquires a particular meaning—that is, a particular data type. We require the user to help us infer the allocation tree for their program, by telling us what *allocator wrappers* and what *sub-allocators* it contains. However, note that the user only needs supply information about allocators over and above the "standard" allocators such as malloc()—knowledge of which is supplied by the per-language front-ends (using essentially the same mechanism). Moreover, the user needs only provide signatures for each function; the actual structure of the tree, in the sense of which allocators nest inside which others, is inferred dynamically from the containment of allocations inside other allocations.

Allocator wrappers A further complication is of allocator wrappers. These have the appearance of custom allocators, but do not actually do their own (nested) allocation. Rather, they simply delegate to a particular underlying allocator, with some change to its interface semantics. For example, many C programs define xmalloc() which delegate sto malloc() but terminates the program on failure, thereby relieving the caller of handling the NULL return value. We similarly require the user to identify these wrappers. This is because it is the caller of a wrapper who sizes the allocation, hence determines the data type; wrappers are effectively a (simpler) special case of sub-allocators.

Allocation sites Allocation sites are simply addresses of instructions in a compiled binary. These are invariably call instructions (as a consequence of our "procedurally abstracted" assumption, which is borne out very consistently in practice). Our allocation site metadata is simply a mapping from these allocation sites to the data type they allocate. Language-specific techniques are in general necessary to determine this type, so it is the role of the language front-end to supply this information. For languages with typed allocation primitives (such as C++'s new) this is manifest in the source code, and sometimes also in the binary code. In other cases, we are not so lucky; we describe how it is done in the particularly tricky case of C in §5.2.

Allocation site information as debug info Logically, we consider allocation site information to belong in debugging information. Allocation, as an event, is something that a debugger user might want to be informed about, to set break points on, and so on. Moreover, tracking the allocation sites of memory objects, much as libcrunch does, could be usefully done inside a debugger too, or inside other consumers of debugging information—particularly profilers. As such, our allocation site information would more properly be output by the compiler rather than by our libcrunch front-ends. e note that many compilers already include foundations for this support, such as gcc's __malloc__ attribute and __alloc_size__ attribute (the latter signposts allocation function signatures; the former denotes that the pointer returned is unaliased). Therefore, while our prototype implementation creates allocation site metadata in our own format, outside the compiler, in the future we hope to engage with the DWARF debugging standards body and compiler authors to generate this information where it logically belongs.

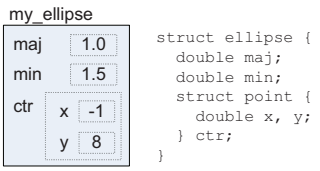

Figure 6. A simple ellipse data type in C and diagrammatically

4.2 Data types in debugging information

If allocation sites are denoted simply by addresses, how are data types denoted? We observe that a ready-made, flexible and language-independent model of data types is already available in debugging information formats. We focus on DWARF, the *de facto* standard on contemporary Unix platforms. Fig. 6 shows a simple ellipse data type in C. Fig. 7 shows its representation in (a textual rendering of) DWARF. Important observations are that the layout is described right down to the binary level, and that every related source-level data type is included. This goes right down to the size and encoding of primitive data (here float meaning the architecture-native floating-point encoding). A stack machine language is used to encode the location of fields relative to the base address of the containing object (here simple offsets).

DWARF is not optimised for speed of access. It is also not deduplicated, in the sense that each compilation unit includes its own copy of any data types that it may have in common with other compilation units. (Some facilities for deduplicating are provided by the format, but they are not consistently used.) We postprocess debugging information to generate records called uniqtypes, of which the generated by our ellipse are shown in Fig. 8. The important features are that a simple, fast in-memory representation is used (each uniqtype is an instance of a C struct uniqtype, compiled into a shared library), irrelevant details are discarded (while retaining important relationships between different types, which in this example include containment and field offsets—note how ellipse refers to point, which both refer to double, and field offsets are stored next to these references), and that identical data types are merged. This allows a simple "exact match" test on two types to be a simple pointer

```
2d: DW_TAG_structure_type
    DW_AT_name          : point
39: DW_TAG_member
    DW_AT_name          : x
    DW_AT_type          : <0x52>
    DW_AT_location: (DW_OP_plus_uconst: 0)
45: DW_TAG_member
    DW_AT_name          : y
    DW_AT_type          : <0x52>
    DW_AT_location: (DW_OP_plus_uconst: 8)
52: DW_TAG_base_type
    DW_AT_byte_size     : 8
    DW_AT_encoding      : 4 (float)
    DW_AT_name          : double
59: DW_TAG_structure_type
    DW_AT_name          : ellipse
    DW_AT_byte_size     : 32
61: DW_TAG_member
    DW_AT_name          : maj
    DW_AT_type          : <0x52>
    DW_AT_location: (DW_OP_plus_uconst: 0)
6f: DW_TAG_member
    DW_AT_name          : min
    DW_AT_type          : <0x52>
    DW_AT_location: (DW_OP_plus_uconst: 8)
7d: DW_TAG_member
    DW_AT_name          : ctr
    DW_AT_type          : <0x2d>
    DW_AT_location: (DW_OP_plus_uconst: 16)
```

Figure 7. DWARF debugging information for a simple ellipse data type

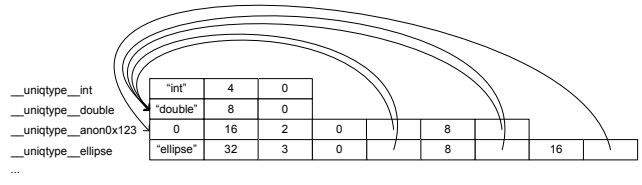

Figure 8. Our run-time representation (slightly simplified) of the ellipse data types and those it uses: pointer arcs represent containment, and the fields preceding pointers are the offsets of the contained subobject. FIXME: point, not anon.

comparison. We will discuss various implementation details in §6.2.

Our reification is pushed right down to the object code level: each data type is a named symbol definition, using a simple human-readable naming convention (albeit one where collisions are theoretically possible). This means it is possible to look up reified data types by name at run time (using dlsym()) or from a debugger (using the debugger's unmodified symbol lookup). Data type synonyms (typedefs in C) are rendered as alias symbols, so all synonyms for a data type, and the data type's symbol name itself, reference the same object in memory. Each symbol name includes a "digest code" which is computed from the data type's definition, allowing disambiguation of name collisions.

DWARF contains information on the stack frames of a function, but unfortunately does not unify these with description of data types. Rather, each function (or "subprogram" in DWARF terminology) contains descriptions of where its actual parameters and local variables are located,

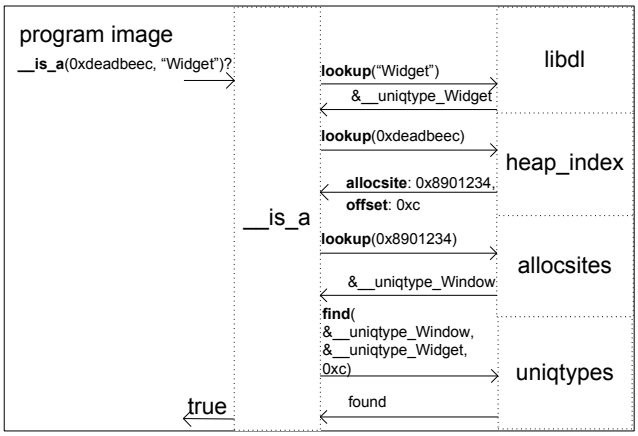

Figure 9. A high-level view of the check dispatch process in libcrunch. FIXME: change to match git example, not widget/window. Get rid of libdl.

in the form of stack machine expressions in terms of the virtual address and the logical "frame base" (an arbitrary stack address stable for the duration of the function activation). We postprocess this information into uniqtypes so that pointers into the stack can be handled as uniformly as possible with other cases. Each subprogram yields zero or more uniqtypes, each valid for a certain range of addresses in its code; this reflects how use of the stack varies as variables come in and out of scope (and as a consequence of compiler optimisations). Some functions' arguments and locals live entirely in registers, in which case the function will generate no uniqtypes—but since registers are not address-takeable, we will never be asked to dispatch a check on a register located value. We describe some details of this process in §**??**.

4.3 Outline of libcrunch execution

Returning to the example code in Fig. 1, we can now piece together how checks are dispatched by libcrunch, as summarised by Fig. 9. Each check arrives as a call to a query function, typically __is_a, with two arguments: the object to check, and the "tested-for" data type which it is expected to match (where the precise semantics of "match" depend on the query function). Rather than the string representation of data types shown in Fig. 1, the function is passed a pointer to a uniqtype. We do a case split on the pointer to discover information about its *allocation*: is it in static, stack or heap storage? A collection of *index* data structures map different storage regions to different; Fig. 9 shows the heap case, but each case is similar: the index returns the base address a value representing the entire allocation (here its allocation site, i.e. the address of the call instruction which made the leaf-level allocation invocation). A second map translates the allocation site into the uniqtype for the whole allocation; logically these could be merged, and an optimisation in fact does so (§6.3), but logically they remain sepa-

rate. Finally, the __is_a implementation searches through the uniqtypes, starting from the allocation's uniqtype, for an instance of the tested-for data type at the relevant offset. If found, the check succeeds, else it returns failure to the caller. Although assert() is one possible reaction to a failed check, in practice we have found it more useful to warn and continue execution, much like Memcheck and similar tools; we describe the instrumentation done by our C front-end in more detail shortly (§5.1).

Note that in cases where only part of the program has been compiled using libcrunch's toolchain, we might have to abort some checks because we find that certain addresses cannot be classified (typically because we lack allocation-site metadata for those addresses). This happens particularly in cases where pointers flow from an (uninstrumented) library to a client which attempts checks on those pointers. These failures are counted and summarised at the end of execution, but do not generate any warnings as they occur.

5. Dealing with peculiarities of C

Our C front-end consists of a compiler wrapper, crunchcc, written using the CIL framework [**?**]. It contains two distinct passes: one to dump the information about heap allocation sites outlined in §4.1, and another to instrument code with calls to libcrunch to check for type-correctexecution. We begin with the latter.

5.1 What to check?

Our C front-end checks *pointer casts*, by default using libcrunch's __is_a test: the pointer points to an instance of (nominally) the given data type. We discard type qualifiers (const and volatile) and check only unqualified ("concrete", in our terminology) target type. Largely this is because const in C is not a property of allocations, but ha sa more fine-grained use: to restrict what access rights a particular pointer confers on its user. We discuss type qualifiers further (§8.1).

C also allows implicit strengthenings of void *, which we instrument just as if a cast had been inserted.

We choose not to instrument operations such as pointer arithmetic and indexing, since these are the domain of memory-correctness tools, which we deliberately choose not to duplicate (§8.1). We mentioned earlier (§2) that we attempt to provide "type-correctness assuming memory-correctness".

We believe that instrumenting casts and implicit strengthenings is sufficient to provide the guarantee of "type-correctness assuming memory-correctness". The reason is that a C program without pointer casts (or implicit strengthenings) could only perform type-incorrect access following a memory violation or following an incorrect use of unions. We discuss unions shortly. (We are working on a semantics of C that will allow us to state and verify this property more formally, but for the moment this remains future work.)

```
len = sizeof (struct dirent);
/* ... */
entryp = malloc(len);
```

Figure 10. An occurrence of sizeof distant from the allocation

5.2 From untyped to typed allocations

As described in §4.1, allocation sites are those places in code which select a data type and call an allocator (at leaf level) to hold one or more instances of it. In C code these are either calls to allocation functions or calls to wrappers of allocation function (like xmalloc()).

We inspect source-level allocation sites to find out which data type they are making space for, by analysing their use of sizeof. In simple cases this amounts simply to observing what data type provides the argument to the sizeof operator inside the allocation function's size argument. Allocation functions are specified in an environment variable LIBCRUNCH_ALLOC_FNS (seen in Fig. **??**) with a simple signature descriptor which distinguishes the size argument.

(The signature is also used to generate stubs at link time to interpose on these allocation function; see §6.1.)

To deal with code like that in Fig. 10, where the size computation is not done directly in the argument expression to the allocation call, our front-end performs an intraprocedural flow-insensitive analysis on each function containing an allocation site: a sizeof expression propagates its sizeofness (i.e. the data type whose size it denotes) to other expressions computed from it by arithmetic. The sizeofness which reaches the allocation function call's size argument determines the data type associated with the allocation site. (Recall that if the caller computes the size, then the caller is an allocator wrapper, so we would be doing this analysis in the caller. Hence an intraprocedural analysis suffices, except in the rare case of *size helpers* which we consider in §5.6.)

This sizeofness propagation is effectively a special case of dimensional analysis; we can think of a value's sizeofness as denoting its "unit" much like a physical dimension. The arithmetic done on sizes is usually multiplication, to create arrays. Addition of (dimensionless) padding bytes also occurs; this does not affect the dimension. Addition of another sizeof value also sometimes occurs; this amounts to creating a composite data type, effectively an implicit struct. We can handle this so long as the order of addition reflects the order of layout of the elements in memory (even though the reverse operand order could be used for the size calculation, since addition is commutative, but this would be very confusing for other programmers to read). FIXME: implement this bit.

More properly we should also analyse use of offsetof, to deal with examples such as that in Fig. 11 which is taken from the Linux manual page for readdir(), where the last

```
len = offsetof(struct dirent, d_name) +
        pathconf(dirpath, _PC_NAME_MAX) + 1
entryp = malloc(len);
```

Figure 11. Using offsetof plus padding to simulate a statically unknown sizeof

```
void *p = ...; // acquire a pointer-to-void somehow
*(char**)p = "Hello, world!";
```

Figure 12. A multiply-indirect cast

```
float *pi;
void *p = &pi; // acquire a pointer-to-void somehow
*(char**)p = "Hello, world!"; // if we tolerate this write ...
float j = *pi; // ... this line makes a type error without a cast!
               // -- will read pointer bits as float bits
```

Figure 13. A multiply-indirect cast

structure element, a character array, is resized according to the value returned by pathconf(). FIXME: implement this, then say as much. NOTE that we need to mark it as a variable-length singleton, to avoid confusion if length reaches twice the struct size or more.

5.3 Storage invariants and multiple indirection

We claimed at the start of this section that after ruling out memory bugs and misuse of unions, instrumenting casts is sufficient to catch type errors. Multiple indirection brings some cases in which it is difficult to see, at first glance, whether this is true. Fig. 14 shows a simple piece of code. When should the cast to char** succeed, and when should it fail?

We take a deliberately strict approach: this cast should succeed if and only if the memory pointed at by p was *allocated as* a pointer to char. If it was allocated as some other kind of pointer, our inserted __is_a() check will fail. (Note that "allocated as" is the usual meaning of __is_a(); here we are simply noting that there is no special relaxation for pointers.)

The rationale is that by enforcing the declared contract of a pointer in this way, we maintain the property that type errors can only occur at cast sites (or following previously check-failing casts), so instrumentation of other pointer accesses is superfluous. If we were to allow a greater degree of sloppiness into what pointers could be written into what storage, a subsequent read or write *not* using a cast might nevertheless cause a type error, and we would be forced to check *all* uses of pointers, greatly increasing run-time overhead. Fig. 14 shows an expanded fragment based on the previous example and exhibiting this behaviour.

It turns out that we observe surprisingly few violations of multiply-indirect pointer contracts in real code, and the overwhelming majority of the legitimate (non-buggy) ones fall into a small number of special cases which can be handled individually. We consider three such special cases in the next subsection. Moreover, violations of the rule are almost always invalid C, at least under C99 and later standards, owing to the aliasing rules based on "effective type", which we discuss in §5.6.

5.4 Tolerating sloppiness

Some styles of C programming are sloppier than others—including, as we just saw, regarding how strictly pointer values are contained within appropriately-typed storage.

These are all policy issues which remain isolated from the core of libcrunch, but our front-end provides some programmer-facing flexibility for eliminating false positives which would otherwise originate from certain common sloppy usages.

Equi-sized types Sometimes when allocating heap objects, the programmer applies sizeof not to the data type that will actually be stored, but to a data type which (he believes) has the same size. A common case is allocating an array of pointers using n * sizeof(void*) even when the pointers will be used as int* or some other type. (This is, strictly speaking, an invalid practice, since void* is not obliged to have the same width as any other pointer type.) We let the programmer work around this by specifying lazy heap types: the type of an allocation may be overridden by the first *libcrunch*-dispatched check on that object. In other words, the type of the allocation may be changed, once, at the time of the first check; after that it must be consistent. Typically this first check happens immediately after the allocation (e.g. int **p = malloc(n*sizeof(void*));, where our front-end instruments the implicit strengthening occurring immediately before the assignment).

Structural matching using __like_a Some data types are designed to be treated structurally. For example, the Berkeley sockets API's generic sockaddr includes padding bytes which specific networks' address structures, like IPv4's sockaddr_in, can "fill in" without changing the structure's length. Use of these padded structures tends to accompany equi-sizing: one might heap-allocate an array of sockaddr_ins using sizeof (sockaddr). In order to go *backwards*, however, i.e. to cast a sockaddr_in * to a sockaddr *, we cannot use __is_a, but must resort to a different style of checking: __like_a. The user can request this by setting LIBCRUNCH_USE_LIKE_A_FOR_TYPES=sockaddr in their environment, meaning all casts whose target type is sockaddr * use the more relaxed __like_a check. This "unwraps" the allocation data type and checks that each corresponding pair of fields satisfies __is_a. Fields typed as char arrays are treated as padding. Note that this unwrapping process goes down only only one level in the subobject containment tree: __is_a on the constituent elements, modulo tolerating padding. A "full unwrapping" would be *physical matching* [**?**] (i.e. checking only the leaves in the

```
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!
```

Figure 14. A cast yielding a write-correct but read-incorrect pointer: the cast yields a pointer which is safe to write any pointer through, but which is not safe to read from

subobject trees). We have not currently implemented this style of checking, since we suspect is more permissive than any real application needs. Again, we note that even the use of sockaddr involves aliasing that is not permitted by the C standard; to avoid compiler optimisations taking advantage of this undefined behaviour, such code must be compiled with special options such as -fno-strict-aliasing.

Read- or write-correct pointers Consider the code in Fig. 14. Since the pointer p was not allocated as struct foo *, the cast will fail, even though the code is perfectly correct. This is a rare example of a contract-violating multiply-indirect cast, and arises because the pointer will only be written through (and any pointer is a void*, so may be written correctly), and not safely read from (since the storage may contain any pointer, not only struct foo *).

We do not currently implement a relaxation which would help in this case, but one approach would be to speculatively tolerated strengthenings of void*-allocated pointers *for writing only*, by issuing (from the cast) a piece of no-access memory onto which reads would generate a failure and writes would propagate to the underlying pointer. Managing the lifetime of this extra piece of memory is tricky: it must last at least as long as the underlying pointer object, so in the case of stack memory would probably need to be freed by hooking the on-stack return address of the containing frame.

The apparent dual case involving read-safe, write-unsafe pointers turns out to be even less common, because of the distinction between rvalues and lvalues in C. An allocation of one or more int*s, say, would not need to be address-taken and cast to void** in order for a distant function to read a single void* from it; in most cases one would simply pass a void* directly. We might still see a cast in the case of an allocation of multiple int*s being address-taken and cast to void** for reading multiple opaque pointers, in which case a dual technique could be applied. However, again we note that void* and int* may have different representations, so doing these casts at the deeper level of indirection is not valid C; standard-compliant code must instead convert each pointer to and from void* individually.

Signed and unsigned slop It is common sloppiness in C to access values using signed and unsigned variants of the same-width integer type. In most implementations, half of the bit-patterns shared between these types have identical

```
/* (generic) library code */
void register_callback (void *(*fn)(void *));
void trigger_callbacks (void *arg);

/* client code */
int *f(double *arg) { printf ("Saw␣%ld\n", *arg); }
// ...
register_callback ((void*(*)(void*))&f); // cast to more-generic type
double myval = 42,
trigger_callback (&myval); // f receives a double*
```

Figure 15. C code casting a callback pointer to a more generic argument type

meanings (the nonnegative half), so this is not necessarily incorrect. By default we consider this mix-and-match to be a likely bug or at least a "code smell", so we leave such casts to fail. However, the programmer can request to disable this, either altogether or only in the context of __like_a checks (which are naturally sloppier) using an environment variable. Our data type representation (refer back to Fig. 8) includes in each signed or unsigned integer data type a pointer to its signedness-complement, making it efficient to match them interchangeably. FIXME: implement the option to enable this.

5.5 Casting function pointers

Casts of function pointers raise a similar problem to multiply-indirect pointers: a cast to a function pointer creates a capability on which future accesses can proceed *without* further cast operations. Therefore, since we check only casts, certain reasonable and necessary casts of function pointers might cause future type errors to occur without checks in place to detect them.

Consider the code in Fig. 15. A client wants to register a callback, which will later be called with a pointer argument of the client's choosing. This pattern is very common in C code. Since the client will supply the argument, it can be any type he chooses; he selects a function which expects a pointer to double. However, the choice of argument type is not known to the (generic) registration code, so this must accept the more liberal type of void*, meaning that buggy generic code might subsequently cause *any* pointer to be passed on the call through the function pointer, *without* a cast being there to check this.

The most precise solution would be to generate wrapper functions for each (static instance of) casts of function pointers, to do the necessary checks on argument and/or return values. For example, we would instrument the code in Fig. 15 by passing instead the wrapper function, shown in Fig. 16.

In order to be compositional, this requires closure creation, since the wrapped function pointer might not be the statically-known &f but might be some other function pointer (perhaps the address of another wrapper, and so on). A further complication is identity: if the wrapper pointer is

```
// to be returned from a cast (void*(*)(void*))&f
void *f_wrapper(void *arg)
{
    assert(__is_a(arg, "double"));
    int *ret = f((double*)arg);
    // no return check is necessary
    // -- caller-side weakening is okay
    return ret;
}
```

Figure 16. C code casting a callback pointer to a more generic argument type

```
// depending on input, can generate various datatypes
// ... so returns void*
void *generate(void *input);

// specialised code wants a generator of ints
void do_with_generator(int *(*)(int *arg));

// client casts
do_with_generator((int*(*)(int*)) &generate);

// instrumentation must pass a wrapper
int *generate_wrapper(int *input)
{
    // we always get an int*, so no argument check needed
    void *ret = generate(input);
    assert(__is_a(ret, "int"));
    return (int*) ret;
}
```

Figure 17. C code casting a callback pointer to a more generic argument type

cast back to its accurate type, it should compare equal with a pointer to the unwrapped function. We therefore need to detect that a function pointer is a wrapper, and "unwrap it" if a sequence of casts has turned it back into its original type.

A similar case occurs with casts which strengthen the return type of a function pointer: a wrapper needs to be inserted which checks the return type. Fig. 17 shows an example of a polymorphic generator function being passed to a client which wants to generate integers only. Passing it to the client entails a cast which strengthens the return type

Our implementation doesn't currently implement these wrappers. Rather, we work around the problem by a slightly less efficient means: we check *all* function pointers at *use* time, checking that each pointer argument satisfies `__is_a` for the argment type specified in the actual pointed-to function's signature, and similarly for the return type. FIXME: implement this bit too.

5.6 Compromise cases

Difficult-to-find sizeof Our intraprocedural sizeofness analysis is incapable of classifying allocation sites which use helper functions to compute the allocation size. (Note that this refers only to the case where an allocating function calls a helper to compute the *size*, but not to perform the allocation—if the latter were the case, this would be an allocation wrapper, and is handled in the usual way.) In our experience, size-helper functions are rare; our prototype does not handle them. In our experience, helper macros are much more common than helper functions, and these are handled naturally since our sizeofness analysis happens after preprocessing.

Reified type descriptors A second possibility is that the size is computed at compile time and stored somewhere, then read (perhaps indirectly) at allocation time. This is a rare pattern in C code, but does occasionally emerge (one instance being the perl codebase, to be discussed in §7.1.1). The object into which the size is stored is We handle only the case in which these objects are statically allocated (which seems likely in general, and matches the one case we have seen in practice) and serve as some kind of descriptor of the data type whose size they embed—typically contining a virtual function table. In other words, these objects are run-time reification of data types. Our approach is to allow the programmer to specify the mapping from *these* reified data type objects to *our wn* reified data type objects, using an environment variable containing pairs of symbol names. This generally also requires the user to reclassify one or more higher functions on the stack to be allocation functions, such that the top-level allocation wrapper function is one which receives not a size but a descriptor pointer. For example,

PASTE PERL EXAMPLE HERE

Unions Our front-end includes only a very liberal kind of checking on unions: it considers a union instance to simultaneously be an instance of each of its arms' data types, such that `__is_a` will pass for any of these types. This contrasts with the C standard, which states that "when a value is stored in a member of an object of union type, the bytes of the object representation that do not correspond to that member but do correspond to other members take unspecified values". Therefore, it may be incorrect to access a larger member, and in general we would need to track which member of the union had last been written. However, it does not contrast with all *uses* of the union construct in C. We are aware of three cases. The first is where a union is discriminated explicitly by a value stored somewhere, invariably nearby, such as a field in an enclosing struct (as in BSD's `getifaddrs()` call, whose arguments include such a union-containing struct, `ifaddrs`) or an accompanying argument (as in POSIX's `sigqueue()` call). We could potentially model these discriminants in our `uniqtypes` and write a checking function that was aware of them (noting that DWARF can express discriminants even though C cannot, using features intended for variant records in Pascal-like languages). The second is where a union is discriminated temporally, such that the surrounding program knows which arm of the union is valid at which points in the union instance's lifecycle. One example is the `int86` call in some C libraries on x86 platforms, in which input registers and output registers are passed in a union such

that the input arm is valid before the call and the output arm is valid afterwards. This is harder to deal with, but could be done by instrumenting the client code directly. The third is where all arms are valid simultaneously; one version of this is the infamous "fast inverse square root" function, which, when translated to modern C [Eberly 2010], uses a union of float and int to provide access to a floating-point number's bit-pattern simultaneously as an ordinary float value and as a 32-bit int. Note that *address-taken* union arms are a particular problem, since they stymie attempts at union-specific source-level instrumentation (by making it statically undecidable whether a points into a union). A proper treatment would force us to dynamically check every pointer use, in case it pointed into a union at a currently-invalid member's type. However, use of address-taken union members violates modern C's strict aliasing rules (the fast inverse square root's "modern C" rewrite accounts for this, by instead taking the address of the whole union) and is rare in any case.

The ambiguity of char We don't check casts to char. The reason is that char is the data type used for bytewise access to opaque, uninterpreted ("untyped") memory, *and* the data type used for access to character data. Checking that all such memory really was allocated as char would introduce a large number of false positives. 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 to library interfaces.

Effective types When designing high-performance instrumentation, it is very desirable not to interpose on memory accesses, since this brings high run-time overheads. Unfortunately, the C standard is specified such that were we to follow it precisely, we would be forced to do so.

> The *effective type* of an object for an access to its stored value is the declared type of the object, if any. If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value.

In other words, a C programmer is allowed to write to a heap object (stack and static objects have declared types) using any lvalue, and the memory will take on the type of that lvalue. Our allocation-based model is stricter: the type of a heap object is decided at its allocation[6], and cannot be changed except by reallocating the memory. This stricter model allows our analysis to avoid trapping memory accesses, but does not create false positives in real (non-pathological) C code.[7] Another example of how we are more restrictive than the standard's "effective types" is the treatment of interchange of signed and unsigned integer types mentioned previously (§5.4): whereas the C standard says that this is allowed, we treat is as an error unless explicitly requested.

memcpy(), memmove() and characterwise copying The C standard states that an object's effective type is propagated as the bytes are copied around (into memory with no declared type).

> If a value is copied into an object having no declared type using memcpy or memmove, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.

Supporting this propagation in our system is possible in the case of memcpy() and memmove()—we reassign the target heap chunk with a new data type originating with the copied-from memory. In general, however, doing so might require us to synthesise new uniqtypes at run time, since we are allowed to copy only *part* of an object, which may span adjoining subobjects but stop short of including the whole of the parent object.[8] In the case of characterwise copies, since these are very difficult to detect dynamically, and to propagate effective types on every characterwise write to the heap would be inefficient, we choose to force the user to refactor their code to use memcpy().

6. Implementation

Our implementation is for Unix machines, currently limited to GNU/Linux running on the x86-64 architecture, but without particular obstacle to ports to other Unix platforms or other architectures.

It consists of three parts:

- language front-ends, currently only CRUNCHCC;
- generic compile- and link-time tools to support allocation and data-type metadata;
- the libcrunch runtime itself.

We describe each of these in turn.

[6] . . . or soon thereafter, in the case of the relaxation we described in §**??**

[7] One could imagine a libcrunch front-end which *did* intercept such writes, either using memory protection traps or dynamic recompilation or by writing a handler for failed __is_a checks. Such a handler could reallocate the memory and continue, using a special allocation function but much like any other allocator described to libcrunch. Quite apart from additional overhead, we consider such "on-the-fly" re-typing of an allocation, even though it is strictly speaking valid C, to be a very likely indicator of a bug!

[8] A defect report made against an earlier version of the standard clarified that this behaviour is expected. http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_219.htm

6.1 crunchcc

crunchcc is a simple Python script which runs the host C compiler. Our analysis and instrumentation passes are written in CIL [**?**], so we delegate to CIL's cilly compiler wrapper for actually running the compiler.

We somehat modify the command-line compiler options, to enable subsequent parts of the toolchain. In particular, we enable debugging, enable break pointers (which speeds up unwinding), and always emit functions in their own section, by passing -ffunction-sections to gcc (to help ensure interposability of allocation functions even in awkward cases).

We run two CIL passes: the first doing allocation site analysis and the second doing instrmentation of pointer casts.

The allocation site analysis proceeds broadly as described in §5.2. We output a plain text file for each compilation unit, consisting of records describing the line and column number of each allocation site (i.e. call to a named allocation function) and the type that our analysis inferred was being allocated. In order to classify indirect calls to allocation functions, we also output such a record for an *indirect call site* whose signature matched an allocation function.

Each instrumented cast is replaced by a call to an inline check function, then the original cast itself. The inline check function (usually __is_aU) handles common trivial cases of pointer casts, such as checks on null pointers. Casts to char* or void* are not instrumented since they would always succeed. The main check function, __is_a_internal(), is defined in libcrunch (see §6.3.

To support the optional usage of libcrunch, in which checks only occur if libcrunch is loaded, we also subtly modify the linker options on the compiler command line. When building shared libraries, we link a no-op stub implementation of our check functions. When building executables, we simply add a dynamic linker dependency (DT_NEEDED in ELF terminology) on libcrunch to the output object. (The no-op stub approach is more transparent, and we would use it for executables too, but unfortunately preloaded libraries do not override definitions in the executable, unlike definitions in later-loaded shared libraries.)

A challenge in the C front-end is that during instrumentation, we cannot yet map source-level C data types to their binary-level uniqtypes by name. The reason is that the compiler has not yet run, so has not yet selected the representation of the data types concerned. Among other things, this means that the digest code (§4.2) and hence symbol names fo C-level data types are not yet known. There is also complexity from the fact that in C, many names exist for the same primitive type (signed int, int, etc.), but the exact equivalence classes between these are copmiler- and architecture-dependent (e.g. long int and long long int are the same on some architectures but different on others). To circumvent this, we force compilation to proceed via intermediate relocatable object files (.o), even if the original command line was a "one shot" compile-and-link invocation. The instrumented code declares its reified data types as extern, using a source-level name, generating additional undefined references in these output objects. These source-level names are also visible in the debugging information generated by the compiler, which, crucially, also includes the *definitions* for the various like-named data types (i.e. the information which was not available at instrumentation time). After each object file is generated, we fix up these undefined references using a separate language-agnostic link-used-types tool (described in §6.2).

Link-time intervention All allocation functions mentioned by the user or known to the language front-end are subject to *stub-generation* and link-time interposition. We use the signature information for each allocation function to generate a wrapper which sets some thread-local variables describing the allocation site (primarily the return address, but also the size being allocated), then calls the original allocation function. These wrappers are inserted using the GNU linker's –wrap option. Some complications emerge when doing this interposition within a single object file, in which scope the allocation function is a *defined* symbol that is also *referenced* from other places in the object file. The –wrap is not effective since it only alters how *undefined* symbols are resolved. We use a specially patched GNU objcopy[9] to *unbind* the definition from its references, at which point references are undefined symbols which can be wrapped in the usual way. We also "globalize" allocation functions, since in some rare cases they are realised static functions, for similar reasons.[10] We also have to interpose on deallocation (free()-like) functions for sub-allocators, so that metadata can be unindexed at the right *level* (see §6.3).

6.2 Allocation utilities

We generate and propagate metadata as code is compiled and linked.

objdumpallocs and merging The language front-end generates a text file alongside each preprocessed source file containing allocation site information inferred from the preprocessed source code (§5.2). We merge this with binary-level information extracted by disassembling the output binary searching for calls to allocation functions and (all) indirect calls. Using the debugging information (which the language front-end ensures is generated, e.g. by enabling it in the compiler options), we map these instructions to their source file, line and column numbers, hence locating the source-level allocation information. The source- and binary-level analyses do not always agree precisely on which source line a given allocation call occurs on, so our binary-level analysis outputs a pair of source code lines denoting the interval in which the call instruction lies. From this we found

[9] from binutils, http://www.gnu.org/software/binutils/

[10] FIXME: mention -ffunction-sections?

it straightforward to robustly merge the source- and binary-level information. Indirect calls observed in the binary but which did not have allocator-like signatures (hence generated no record during source-level analysis) are naturally discarded during this merge. From the merged information we generate a binary allocation metadata file in the form of a shared library. with a single data section. This contains an array of fixed-length allocation site records, include a symbolic reference to the uniqtype being allocated. This reference is satisfied by data-type definitions in the types library which is generated separately.

dumptypes and friends The tool dumptypes is run after a linked binary is produced, simply generates a re-encoding of its DWARF information into our in-memory uniqtype format, in the form of a C output file which is then compiled into a shared library. A uniqtype is a C stucture consisting of a name, size and a variable-length array of contained subobjects (offsets and pointer to their uniqtypes). Slight variations on this are used to encode base types (which embed a pointer to their signedness complement, if any, instead of subobject types), arrays (which embed their element type), pointer types and so on. A variation on the same tool is link-used-types, which is called by language front-ends to fix up undefined references in object files; this emits only the used data types and those they depend on, then invokes the linker (in relocatable-output mode) to produce a single object file. We dump the binary's stack frame layouts at the same time, along with a table describing the range of program counter values for which each layout is valid. Since most working storage is kept in registers, including many local variables and actual parameters, we found that most functions generate only one frame layout or a small handful, but a few large functions have been observed to generate between 50 and 250 (typically as a result of inlining).

Keeping uniqtypes unique We produce -types.so objects recording every data type in each output binary. We also link (in link-used-types) the same information into instrumented user binaries, since they wish to refer to these in checks.[11] We use two linker techniques to ensure that despite this apparent duplication, a unique definition for the same data type is used at run time. The first is "comdat" linking, as used for linking out-of-line copies of C++ inline functions. Each uniqtype is output in a separate ELF section marked with the COMDAT flag and tagged with the uniqtype's symbol name. If multiple sections with the same tag are linked by the compile-time linker, all but one will be discarded.This allows link-used-types to insert a copy of any referenced uniqtype into each object, without fear of causing multiple definition errors at link time. (Note that this choice of section- rather than symbol-level uniquing is a quirk of ELF.) The second is the usual dynamic linker behaviour of global symbols: only a single definition of a global symbol can be active in a dynamically linked program, so if a -types.so object file is loaded containing uniqtypes that have already been loaded (say, because the executable uses struct stat but so does a library), all references to the uniqtypes (including references *internal* to the just-loaded -types.so object will use the already existing definition). This does lead to some "dead spaces" in the mapped library files, where uniqtype records are left unused because they duplicate one loaded earlier. (FIXME: could optimise memory by clumping likely-duplicates together! i.e. reordering uniqtypes) A consequence of this is that dynamic loading falls out nicely: loading new libraries at run time (and subsequently loading their uniqtypes, which we do automatically) does not risk duplicating existing uniqtypes nor require any special work to merge them.

Computing reliable stack layouts We found that the way some on-stack allocations were recorded in DWARF information was problematic. To compute the stack frame layout, we need the address of each actual parameter and local variable as an offset (positive or negative) from the logical *frame base* address. This is simply an arbitrary stack location constant for the whole function's activation (i.e. the compiler is free to choose its position relative to any on-stack storage; it is often, but not always, the value of the stack pointer on function entry). We found that occasionally an actual or local which was stored in a stable stack position (over some interval program counter values) had been described by the compiler not in terms of the activation's frame base, but in terms of some other register (say "rbp + 8") which happened to be holding a stack address (over the same interval). This is fine for a debugger, which can query for the actual register values during execution, and hence locate the value in memory. But it is not fine for us, because our metadata needs to describe stack frame layouts independently of the program's address bindings on any given run. We fixed this by merging the DWARF .debug_info information (the information typically used to locate variables), with additional information from the .debug_frame section (typically used to walk the stack). The frame information describes how one register's value at a given point in a function's execution can be reconstructed in terms of other registers' values. Interpreting fixed-offset relationships as weighted edges in a graph, we used graph reachability to rewrite the .debug_info information for every local and actual to be expressed as a fixed offset from the frame base (i.e. a sum of edge weights) wherever it could be expressed in this way (i.e. wherever such a path existed in the graph, possibly over multiple hops describing how register A is a fixed offset from register B, and so on).

Stack frame coverage We lack information on on-stack temporaries, since the compiler does not generate debugging information for these. Fortunately, in the case of C, tempo-

[11] We cannot let those references dangle until loading of the -types.so object because of how dynamic loading works: previously-loaded objects are not searched for references to newly-defined symbols.

raries never have thei address-taken so cannot be subject to casts. (This is not true of C++, so we would need to force the compiler to generate information for temporaries if we wished to support full stack coverage for C++ code.) We also have no information on on-stack variadic actual parameters; these are rarely address-taken. FIXME: say about how to fix with compiler annotations?

Metadata hierarchy We maintain our metadata in a separate filesystem hierarchy (/usr/lib/allocsites) using make [Feldman 1979], so that the metadata for rebuilt binaries can be rebuilt with a single command.

Aliased uniqtypes A surprising amount of complexity is involved in ensuring that different names for the same uniqtype are handled correctly. In general we create aliases for typedefs and similar aliasing features, and also for primitive types having multiple names (e.g. int, signed int, etc.). Since each symbol name also includes a "digest code" computed from the content of each definition, we can avoid unintended aliasing among like-named definitions that are nevertheless different. We also, however, create "codeless' alias symbols where this is unambiguous, e.g. so that __uniqtype__signed_int denotes the signed 32-bit integer in most binaries (those not defining a conflicting signed int data type) where it is an alias of __uniqtype_05042024_signed_int (its full name including the digest code). This helps users wanting to look up a data type by its symbol name.

6.3 The libcrunch runtime

The role of the libcrunch runtime is to respond to incoming queries, such as __is_a(p, u) ("is pointer p pointing at an instance of data type u?") as quickly as possible. To do so, it must

- maintain a map of the address space sufficient to *classify* any address as *static*, *stack* or *heap*;
- for the heap case, maintain a fast index of heap memory modelling our hierarchical view of allocations (§4.1) and mapping arbitrary heap addresses to their leaf-level allocation sites;
- for all cases, not only the heap case, support fast mappings from static data addresses, stack addresses and heap allocation site addresses to their uniqtype information.

We cover each of these in turn.

Address classification Given a query __is_a(p, u), we perform some quick tests on p which can classify common cases extremely quickly, and then fall back to a more general (and still very fast) method. The quick tests check the pointer against the bounds of the current stack, the executable's data segment and the sbrk()-managed heap. In many cases this is sufficient to yield a positive classification (as stack, static and heap respectively). For other cases we must resort to a structure which maps the address space at page granularity, called the level-0 index. Logically speaking, this associates page numbers (i.e. virtual addresses right-shifted by n bits, where $n = 12$ for 4kB pages) to a *mapping* which is a contiguous region of memory recorded as being stack, static or heap, and corresponds roughly to the structure exposed by Linux's /proc/<*pid*>/maps file [**?**]. We preallocate a fixed-size array of mapping records (currently 1024 of them) in libcrunch's data segment.

Maintaining the level-0 index Rather than continually re-reading from /proc, we exploit the fact that libcrunch is a preloaded library to interpose on C library operations which change the map, namely memory mapping and dynamic loading calls. In some circumstances, mappings change without our seeing them, because our preloaded mmap() does not catch all calls—particularly those internal to the C library. Fortunately, this is limited to anonymous mappings made by malloc(), which we hook using a different mechanism (discussed below). To save space, we merge adjacent mappings of the same file that differ only by permissions, unlike /proc/<*pid*>/maps. However, we split anonymous regions more finely than /proc/<*pid*>/maps, as explained shortly.

Virtual memory Our lookup from page numbers to mapping records is implemented using a virtual memory technique. We map, but do not reserve (i.e. we use Linux's MAP_NORESERVE flag to mmap()), a large array of 16-bit integers—one for every page in the lower 47-bits' worth of address space.[12] Only those parts of the array corresponding to used portions of the address space will actually be committed by the operating system. When a mapping of n pages is created, we grab an unused mapping record from the array in libcrunch and write its index into the n corresponding contiguous locations in the array. Similarly, mapping index 0 denotes "not mapped" and is used when mappings are deleted. This is similar to existing applications of unreserved virtual memory for fast linear lookups [Nagarakatte et al. 2009; **?**; **?**]. Fig 18 shows this arrangement diagrammatically. We will see some more complex uses of virtual memory for heap metadata shortly.

Indexing the malloc() heap Whereas the level-0 index maintains page-granularity mappings, the C library's malloc() manages objects from a few bytes to gigabytes, so we require a different index for these objects (which we call heap chunks). We use the GNU C library's malloc() hooks mechanism to receive upcalls on each malloc()-family API call.[13] The goal of our malloc() index is to map from heap

[12] On the x86-64 architecture, addresses are 64 bits wide, but only 48 of them are defined. User-level addresses lie in the bottom half, and kernel-level addresses in the top half. By indexing the bottom 47 bits' worth, we index all user addresses.

[13] In principle, other hook techniques could be used here on other systems, such as link-time interposition, PLT rewriting, etc..

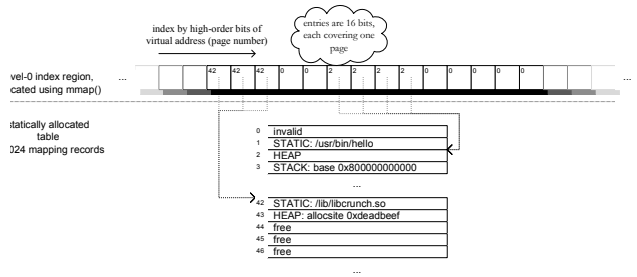

Figure 18. Our level-0 index structure using a large linear lookup in virtual memory

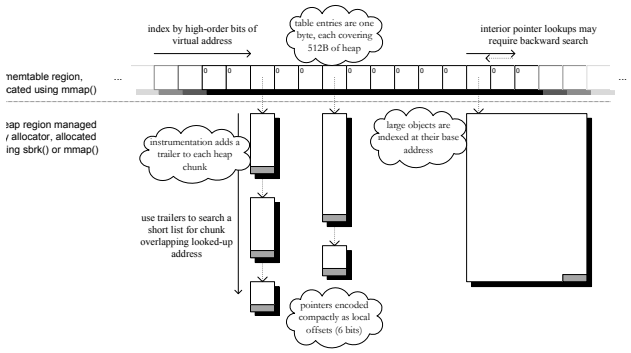

Figure 19. Our heap-threaded memtable indexing the malloc() heap

addresses—not necessarily the *start* address of a heap allocation but any address inside a leaf-level allocation—to their allocation sites. Fortunately, in the case of very large objects (over the "mmap() threshold"), malloc() delegates to mmap(); we detect occurrences of this by inspecting the address returned by mmap(), and push the allocation site metadata directly into the level-0 index's mapping records. (This might require splitting the mapping record currently describing the anonymous region containing the allocation.) To deal with smaller allocations, for sizes between a few bytes and a few hundred kilobytes, we use a novel arrangement called a *heap-threaded memtable* which combines aspects of hash tables and the "large linear lookup" virtual memory technique of the level-0 index.

Heap-threaded memtables Since we need to map from arbitrary heap addresses to their allocation sites, our map must handle range queries of the form "what allocation *overlaps* address p?". We restrict ourselves to the case where allocations (unioned with unallocated space) form a flat partition of heap memory; separate data structure will deal with the case of sub-allocated heap regions (below). A hash table is unsuitable for answering range queries, and also destroys locality (since the hash function's role is to spread nearby keys across the whole hash space). A large linear lookup on its own is unsuitable since any 8- or 16-byte aligned address might begin a heap chunk; and allocating even a single byte for each such address would use up a large fraction of the available virtual address space. Instead we use a hybrid scheme: we group heap chunks in small buckets, through which we thread a doubly-linked list by adding padding to incoming malloc() calls, i.e. by incrementing the requested size. We find that adding an 8-byte trailer to each chunk is sufficient to record *both* the list pointers and our allocation site metadata (a 48-bit address), since we can use a compact pointer representation.[14] Each bucket covers a particular *small* range of chunk start addresses—512 bytes in our case. This means that pointers within a bucket's range can be encoded in only a few bits (5 or 6 bits respectively for 8- or 16-byte chunk alignment). We then index the collection of buckets using a large linear lookup in virtual memory, with one byte for every 512-byte heap region. This avoids using too much virtual address space, but remains fast, since buckets are small (at most 32 chunks for tpical *malloc()* implementations, and typically much less for typical client code, where chunk sizes average 80 bytes). It also has good locality, since each bucket resides within the same 512-byte heap region (a few cache lines). Crucially, it supports range queries: we can search backwards for an object overlapping address p by starting at the heap bucket containing p (found via the linear lookup) then considering preceding buckets in the linear lookup. Since this structure only contains objects of size below the allocator's mmap() threshold, we never have to search back very far even if the lookup fails (e.g. a 128kB mmap() threshold would mean scanning at most 256 bytes of the linear lookup). Fig. 19 shows our heap-threaded memtable diagrammatically.

Advantages over placement-based approaches Another family of approaches to associating metadata with heap allocations is based on *placement*: the allocator uses the allocated address to encode its metadata. The "big bag of pages" allocator [**?**] partitions the heap virtual address space into regions based on the allocation's size or data type, such that a chunk's address suffices to infer its metadata. Our approach is more compositional, more flexible and tends to use less virtual address space. Placement inherently brings a "dominant decomposition" phenomenon since only one placement policy can be used at one time. By contrast, multiple heap-threaded indexes can index the same set of chunks in different ways (by composing trailers and using a separate linear

[14] Trailers are more robust than headers in practice, since they do not alter the alignment of the returned memory. For example, given a caller using memalign() to allocate a 2^k-byte-aligned heap region, we would have to allocate a chunk of 2^{k+1} bytes, i.e. double the size, be sure of having space for a header *and* at least the caller-requested amount of in-chunk space at the right alignment. In this case the header must also encode the "real" start of the chunk somehow, to be able to to correctly free() it. A side-effect of using trailers is that we expect the malloc() implementation to provide the common but non-standard call malloc_usable_size().

lookup). Encoding a large space of metadata values means using a very large virtual address region. Our scheme can index the whole virtual address space using only $\frac{1}{512}$ of the available virtual addresses, and can attach arbitrary amounts of metadata to each allocation. Even our 48 bits (representing the allocation site) would be impossible to encode in a placement-based approach on our chosen CPU architecture.

Indexing deeper allocations We use an alternative scheme to index "deep" allocations—that is, heap allocations "sub-allocated" out of a malloc()- or mmap()-backed allocation, or (recursively) under other such allocations. We are still gathering experience with nested allocators, so have not optimised this case to the same extent as the usual malloc() and mmap() cases. However, we note some points about when and why an application would contain a custom allocator. Programmers tend to use nested allocators to allocate very small objects and/or fixed-size objects, neither of which are handled optimally by a malloc() (although programmer intuition is often wrong about whether savings can be had over a modern malloc() by doing so [**?**]). Certain properties of a malloc()-like interface which our heap-threaded approach relied—arbitrary sizes, relatively coarse alignments and the ability to infer a chunk's size from its base pointer[15]—are no longer properties we can rely on. As a consequence, we may no longer thread or embed our metadata; we must keep it separate. However, we can expect a relatively more consistent use pattern, since a more specialised allocator will not be shared program-wide in the same way that a malloc() is. We use a simple array of metadata records (each currently 16 bytes in our case) sized according to two parameters: the minimum alignment of chunks issued by the suballocator, and a "compression factor" chosen according to the expected average size of allocations in the region. FIXME: is this definitely slower/worse tha the heap-threaded case? It's a little worse for locality and performs worse, not better, for larger objects, but would be good to investigate in practice. FIXME: revisit this section once I've done more with perlbench. FIXME: mention Java: can get away with one bit per object start, because metadata is encoded into data.

Piggy-backing the deep index We allow a relatively small number of "deep allocated" heap regions (currently 63), i.e. regions managed by nested allocators. We piggy-back these onto the memtable by storing a bit-pattern into the linear index that would not be used for a normal bucket pointer (we reserve 63 such values). When we see such a value, we have to consult the deep index instead of walking a heap-threaded bucket. We allocate a fixed-size array (of 63 entries) of records describing deep regions, including their start and length. Each in-use deep region also has a linear virtual memory area associated with it, which logically holds its array of allocation metadata records.

[15] For example, we note that our use of trailers depends on the malloc_usable_size() call to locate the trailer at the end of the chunk.

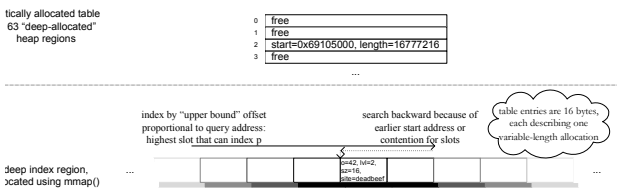

Figure 20. A "deep-level" index structure

Placement in the deep index region We constrain the placement of an allocation's record in a scheme reminiscent of open hashing, but again preserving locality. Given the start address of a deep-allocated chunk as an offset in its containing deep region, we say its "natural" metadata record index is computed by scaling its offset by the alignment *and* the compression factor. So, for a four byte alignment and compression factor 8, an allocation at offset byte offset 1024 would naturally be indexed at metadata record position 32 (1024 right-shifted three times for the compression factor and twice for the alignment). However, compression has created contention: any allocation up to offset 1056 would would share the same natural index, so might already have taken that metadata record slot. We solve this by searching *down* the index space for a free metadata record slot. To handle the case of contention around offset zero, we "wrap around" to the high end of the array. If we chose the compression ratio based on a true (under-)estimate of the average allocation size, we will find a free index in a small number of steps. The expected number of positions we have to seek is proportional to the compression factor (the frequency of unused entries is inversely proportional). Controlling the compression factor therefore allows us to adjust a time-memory trade-off.

Search in the deep index region Our use of upper bounding to place metadata records also allows us to do the necessary range queries. Recall that we wish to search for objects beginning *at or before* an address. We start out search at the natural location, which is the upper bound *both* for an object starting at our query address, *and* for objects starting before it. On a per-region basis we store the "maximum displacement from natural" at which we've stored a metadata record, which gives us a bound on how far to search before giving up. Fig. 20 shows the arrangement diagrammatically.

Other indexes We also maintain three similar tables for allocation site records: to map code addresses to their stack frame layout, to map from allocation sites to their allocated uniqtype, and to map static data addresses to their data types. Each kind of allocation site record is output in the same format in the metadata shared libraries (§6.2), with space for threading a linked list through each record. We then use threaded memtables for these lookups too. As with the heap index, we require the ability to do range queries, so memtables a preferable to hash tables. They are also simple,

fast and avoid dependency on any part of the C library except the mmap() system call wrapper. We fill in the bucket structures when loading the metadata libraries. Each bucket must span no more than 4kB, to maintain the property that buckets don't span different program binaries (which our initialization logic assumes). Currently each bucket spans 256 bytes (making these memtables' linear lookups rather larger than the malloc() memtable's).

Handling stack and static pointers We use libunwind to walk the stack up to a frame containing the target pointer, then use the (saved) program counter of the stack frame to index the allocation site table, yielding a uniqtype representing the frame layout. Static pointers are even easier: we can look up the allocation directly in the stack allocation sites table.

Isolation Currently libcrunch uses the host C library. When building codebases with libcrunch, we instrument only that codebase, and not the C library it links with. A better approach would be to link libcrunch statically against a minimal C library, of which only a handful of used routines would be linked in; this would then allow the C library itself to be instrumented to use libcrunch. (This requires care to ensure that two C libraries can peacefully coexist in the same process.)

Caching optimisation The whole system is very amenable to caching. Once a heap chunk's allocation site has been mapped to a uniqtype, we write the address of that uniqtype directly into the chunk metadata, hence avoiding repeated lookups. FIXME: others once I've done them.

6.4 Implementation status

All the features of libcrunch that we describe in the paper are implemented and working at the time of writing, except for the handful which we have already noted clearly at various points in the text. Our source code is available online at the time of writing.[16]

Thread-safety is the major to-do item in the runtime; we have not yet applied it to multi-threaded programs, although we have had this requirement in mind. Our data structures are simple and very amenable to lock-free programming. For example, our heap-threaded memtable for indexing malloc()-managed heaps reserves the extra bit per list node necessary to implement Harris's lock-free list algorithm [**?**], while our level-0 and deep-level index records can be grabbed and released using compare-and-swap.

7. Evaluation

Since the SPEC benchmarks constitute real codebases of considerable size (FIXME: size numbers), the process of compiling and running the benchmarks was one which yielded considerable practical experience of applying libcrunch to a variety of codebases.

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

We also include details of our experiences with a few still-larger codebases as further case studies.

7.1 SPEC CPU2006 benchmarks as case studies

We took the twelve SPEC CPU2006 benchmarks written in C,

7.1.1 perlbench

perlbench was by far the most challenging codebase to which to apply libcrunch. It is written in a very liberal style, includes its own allocators, and uses a "stored sizeof" approach (§5.6) in which certain subsystems are described with run-time descriptors embedding the size of their private data structures (much like class descriptors in a JVM or similar runtime).

7.1.2 bzip2

bzip2 was mostly straightforward but uncovered some subtleties in our link-time interposition on allocation functions. The code uses address-taken allocation functions, which they are both address-taken (used) and called (defined) in the same file. This yields object files in which references to a *defined* symbol, ans so the linker's –wrap option has no effect (it only affects undefined symbols) and we could not wrap the allocations. We resorted to producing a patch for GNU objcopy which could "unbind" the references from the definition of a symbol specified on the command line, by duplicating the symbol into __ref_ and __def_ "halves". We could then wrap the __ref_ symbol and then use renaming to remove the special names. This logic is in crunchcc and is applied whenever such cases are detected (by inspection of symbol tables).

gcc

mcf This FIXME simulation allocates huge arrays, so placed particular emphasis on our level-0 memory index (§6.3).

milc

gobmk

hmmer

sjeng

libquantum

h264ref

lbm

sphinx3

7.2 SPEC CPU2006 performance results

shamelessly drafty figures for now

name	crunch-time	nocrunch-time	%slower	#checks
bzip2	1.80	1.80	0	1933
mcf	2.71	2.50	8.4	8250423
milc	25.21	9.00	180	84121769
gobmk	11.66	9.82	19	3277616
hmmer	2.32	2.18	6.4	118
sjeng	3.13	3.25	−3.7 (!)	4 (!)

7.3 Additional case studies

7.3.1 git

mapped files – would be great to treat these as just another allocation, using DWARF descriptions of their format to capture their layout... but we ignore these for now

8. Discussion and related work

8.1 Discussion

At this point we take a moment to reflect on the relationship between our system and others with which the reader may be familiar.

Relationship to memory correctness The property that our system is intended to check can be thought of as a complement to memory correctness.

We borrow the design of Memcheck

It also has a hierarchical notion of allocation, in the sense that it supports *memory pools* . Memcheck's support for memory pools requires hints ("client requests") to be compiled in to the program. We can get away without requiring this because we use *link-time* interposition to insert our wrappers.

Relationship to type checkers The most visible specification mechanisms in unsafe languages are static type-checking systems (henceforth "type systems"), often tightly coupled with a language's semantics. (Note that a language's *data abstraction mechanisms*, including the language of *data types* that it can express, is logically separate from its "type system", despite much terminological confusion suggesting otherwise.) Unsafe languages usually have type systems, because static reasoning, when assisted by the programmer, allows elimination of run-time overheads and indirections, such as object headers, packed multidimensional arrays, pointers to on-stack objects, and so on.[17] A key problem in C is that its type language is not expressive enough to specify common invariants. For example, a popular example is the use of generic pointers within embedded linked data structures [**??**]. Here, a lack of polymorphism in the types which may be assigned to the link pointers forces the use of underspecified void pointers. However, even in C++, whose type system offers a form of parametric polymorphism, casts are frequently necessary, because yet more complex invariants are a common feature of real programs. For example, consider a simple length-affixed buffer, perhaps implementing character strings. The invariant relating the length of the buffer to the value of the length field cannot be expressed even using C++'s parametric polymorphism (without statically determining the length of each buffer instance).

[17] BCPL [**?**] is a notable example of an unsafe language with no type system—and is perhaps unique, aside for assembly languages.

Relationship to "type safety" We have so far deliberately avoided using the phrase "type safety", since we consider it to have no widely-agreed meaning, and its use usually causes more confusion than clarity. In particular, it can mean a run-time property —one which our system exists to check— but can also refer to statically verified assurance that type errors will not emerge at run time. FIXME: rant some more? Mention Saraswat again

something about guaranteeing safety under (hypothetical) full instrumentation and memory correct execution, modulo policy. What policy? For C, we believe that checking casts (and implicit strengthenings) is sufficient (backref). We plan to formalise this in future work. Policy is inherently language-dependent (or, for multi-language compositions, a lower-common-denominator is needed).

Relationship to polymorphic or generic data types The combination of generic programming and static type checking, whether by C++ templates [Stroustrup 1997], Java-style generics [**?**], Hindley-Milner style systems such as in Standard ML [**?**] or Haskell [Jones 2003] has led to a parameterised notion of data type in contemporary programming languages. Code which uses such notions is called polymorphic. Our system is naturally amenable to checking polymorphic code—indeed, the primary purpose of void* in C is to accommodate polymorphism. However, our checks tend to occur relatively "late": they show the proximate cause of a failure rather than the root cause. For example, currently a C programmer using our system might be able to check that a list node was a ListNode but not distinguish a ListNode-ofint from a ListNode of double, say. Rather, this "payload-level" type error would occur later, when the pointer to the element data was downcast incorrectly. However, there is nothing in our design to prevent these kinds of checks. We noted in §4.2 how our reified representation encodes *relationships* between data types. The class of abstraction–concretion relationships which relates, for example, a generic ListNode with a specialised ListNode-of-string are very much something that can be encoded in our metadata. Template-based C++ code already generates distinct DWARF information for these two cases, so accommodating the right checks is simply a matter of writing the right checking functions. (Abstraction–concretion already arises in C, in the case of arrays—which may be of known or unknown length, the latter case abstracting former—and pointers.)

Relationship to type qualifiers Adding yet more powerful type systems promises potential solutions. Deputy has added dependent types to C [**?**]. CQUAL adds user-specified nominal dimensions to existing types, and be checked according to user-supplied proof rules [**?**]. [**?**] extends this approach

with a check that typability under the proof rules correctly entail preservation of the intended invariant. Similar systems exist in other languages, such as X10 [**?**] and Xanadu [**?**]. Applying powerful static reasoning to unsafe code seems like a good fit, since static reasoning need not impact runtime performance. However, this comes at a cost: stronger static reasoning can only be achieved when stronger type annotations are applied *throughout* a program. This entails not only annotation effort, but also, some degree of refactoring work to ensure that the program's dynamic behaviour (e.g. the sets of arguments and return values passed and returned at a given function) projects cleanly enough onto its static features (e.g. the function's identity) that a precise annotation (e.g. a function signature) may be used to describe each. Even simple extensions to type systems have this "transitive changes" property, as has long been familiar to C programmers since C90's introduction of the const qualifier: "const poisoning" refers to the chains of modifications that are demanded by the type checker once the const type qualifier is introduced in some (possibly distant) piece of code [**?**].[18] FIXME: mention typestate? type refinements?

Type qualifiers are another way of producing more refined specifications given a data type definition. Certain uses of type qualifiers translate to libcrunch because they apply to allocations. Those that apply to particular *uses* of a value require care, but could be rendered into a slightly modified form. For example, a C function taking an argument using the qualified type const char * does not mean that its target must have been allocated const; rather, it promises not to mutate the target data, regardless of whether other code might be permitted to do so.

FIXME: mention volatile.

Currently, type qualifiers check the local side of the bargain—that the code which consumes the pointer sticks to its no-mutation policy—whereas the flip side, that whatever the client code *does* wish to do *is* supported by the allocation. We might witness this in a hypothetical converse case where we could declare an argument as defining mutable char *. We could then use libcrunch, extended to reify mutable-qualified data types, to check that the characters received do come from an allocation supporting the ability to mutate them, e.g. that it is not passed a pointer to an allocation in a read-only segment. (In this case, memory protection hardware already catches violations, but we could imagine similar properties not enforced by hardware.) However, this is fundamentally different from the per-allocation property: what is allowed depends not only on the object, but on which code is asking.

There is also the issue of *flow of permissions*: use of qualifiers is enforced as values are communicated (through arguments and return values, or by writes to shared storage) from one piece of code to another, intuitively following a "narrowing rule" where the recipient is allowed to demand the same of weaker permission as the sender, but no stronger.

The counterpoint to this is that type qualifiers are notoriously burdensome on the programmer—the "virality of const" was remarked on when it was introduced to C,

In practice, "casting away const" is commonplace in complex C codebases. The "correctness" of such a cast is difficult to define. We could define it only using allocation-level properties, in which case libcrunch would be well-placed to check such casts. We might also decide a more precise style of specification to be more effective: perhaps elaborating on exactly which objects could be mutated by which code (but remaining more relaxed on *how they obtained a reference*—the key inflexibility we just noted about compiler-enforced qualifiers).

In short, our infrastructure is suited for the coarser style of *per-allocation* specification, tending to yield properties which are checkable with little or no added work by the programmer, but not as fine-grained as is sometimes desired.

FIXME: cite CQual, flow-sensitive and flow-insensitive papers, Chin / Millstein

8.2 Other related work

SAFECode SAFECode [**?**] is a system which instruments C code with dynamic checks to enforce a very particular property: that the program stays within the bounds computed by a (whole-program) pointer analysis ahead of time, or the program terminates cleanly. This property is effectively a kind of sandboxing, useful in combination with *static* analyses which consume the outputs of the pointer analysis. It is therefore not, by itself, a dynamic analysis comparable to ours (Our checks occupy a role similar to the static analysis—both assume a kind of memory correctness.) A run-time partitioning of the heap is used to enforce type-based properties for a subset of heap locations. The extent of these properties depends on the pointer analysis. As described earlier (§6.3), our system retains flexibility by avoiding partitioning the heap. Moreover, unlike SAFECode, our system is designed expressly to avoid any whole-program analysis; we are free to compile as much or as little of the system using libcrunch as is required and/or convenient.

Wright—"practical soft typing" thesis; actually mostly theoretical

Loginov & Reps Loginov... Reps—"Debugging via run-time type checking"

—uses a four-bits-per-byte shadowing scheme

—much more heavyweight

—treatment of malloc() is to say "block type is undefined until written to"

—slowdown is 10x–157x

—definitely "for C"

[18] The addition of const qualifiers to the C type system (as opposed to the set of storage class specifiers) is considered a mistake by some commentators. See Ian Lance Taylor's piece on "const", available at http://www.airs.com/blog/archives/428 (retrieved on 2011-12-17).

Shen et al, "Securing C programs by dynamic type checking", IPSEC '06

Burrows et al, Hobbes – big slowdown

Hackett & Aiken FSE'11 "inferring data polymorphism..."

Steffen "adding run-time checking to the portable C compiler" – memory only again

S. C. Kendall, "Bee: runtime checking for C programs". USENIX Toronto 1983 Summer Conference Proceedings, USENIX Association, El Cerrito, CA 1983.

claim in Yong & Horwitz RV '02 "Reducing the Overhead of Dynamic Analysis", that "certain valid program behavior, such as storing the address of a stack variable in a global variable, or storing a pointer value in an integer, casting it back, and dereferencing it, will cause a runtime check to fail"

Runtime Verification community "parameterised monitors"

Variants of C CCured [**?**] is an extension of the C language which supports run-time type- and memory-safety properties— enforced dynamically but performing a large proportion of its reasoning statically in most cases. CCured provides strong guarantees, but at the expense of imposing on the programmer far more than in our approach. It require source-level modifications (FIXME), only provides binary compatibility given modifications to the original program (to turn "WILD" pointers into a more restricted kind) requires "wrapper" specifications for calls to external libraries enforces use of a conservative garbage collector,

CCured also uses an inflexible notion of type compatibility (physical subtyping)

New language designs inspired by unsafe languages, including and Cyclone [**?**] offer languages with similar feature-sets to C but with stronger safety guarantees (respectively FIXME and FIXME). However, they do so at the expense of at least some of the favourable properties of unsafe languages (mentioned in the Introduction): they impose greater run-time overhead, and limit the idiomatic flexibility available to the programmer (examples please). Jekyll [**?**] and BitC [**?**] push a similar strategy further towards "modern" language designs influenced by functional programming, but offer the same problems. New languages often have merit, but require an investment of effort to adopt, so suffer bootstrapping problems. We will prefer solutions which do not constrain the source language.

Optional & gradual typing It is not only performance but also flexibility which drives the continued popularity of unsafe languages. Adding a complex type system in a mandatory fashion greatly impacts flexibility. As noted by other authors [Bracha 2004; Wrigstad et al. 2010; **?**; **?**], more complex types create a more onerous obligation on programmers to *prove* properties of their program, making code slower to write and more brittle to maintain. Moreover, the conservative application of decidable proof languages necessarily rule out programs which are nevertheless correct (i.e. ones in which no type error will occur dynamically). Extending a type system while holding to the usual contract of a type system —that the programmer is obliged to produce a program that type checks (modulo use of any escape hatches such as casts or other unsafe builtins[19] brings a problem of diminishing returns. This has motivated work on gradual [**?**] and "hybrid" [**?**] type systems that support annotations spanning a spectrum of strengths, where static assurances are enabled by stronger annotations, but on occasions where the annotations are too weak to prove a given property, dynamic checks are inserted. So far, however, this work has not yet considered unsafe languages—instead assuming a language with built-in dynamic checking. We consider this line of work complementary to our own: we are extending unsafe languages such that they could be amenable to the properties enabled by gradual or optional typing systems. One possible step along this path, using symbolic execution techniques, is outlined in the next section.

liquid types?

9. Future work

9.1 Combining with symbolic execution

Reifying dynamic checks as assertions makes them "generic properties" of a kind which generic program analysis tool can reason about. One particularly relevant class of tools is symbolic execution engines like KLEE. Our toolchain extensions naturally allow KLEE to search for type errors in addition to the usual (memory, arithmetic, assertion-failure) errors that it searches for.

9.2 Polymorphic types and other abstract checks

As detailed in §4.2, it would be a relatively small extension to add additional *relations* among data types into our reified representation. Doing so could allow additional checks to be expressed. For example, a typical requirement in Java-style generic types is to check type parameters, either exactly or using "wildcards" with bounds. For example, to check on a linked list that the precise T for which List<T> was instantiated is exactly X, or perhaps that it "is a" X. We might want to check the converse: that it is "at most a" X. Doing so efficiently means including a "contained by" relation into our reified representation. This requires some amount of link-time processing, s that when new data types are linked in

An interesting question is how "smart" the reified types should (or must) be, versus how smart the checking logic should be.

A likely path would be to assume that all instantiations of generic data types come with a *contract*

Much like a C compiler erases sizeof, so a Java compiler erases these contracts, and in both cases we must preserve this information by out-of-band means. This fits neatly into

[19] An example of an unsafe builtin other than casts is Haskell's unsafePerformIO.

our existing model of attaching metadata to allocation sites. Whereas the bytecode might allocate a List, our metadata would record that the user allocated a List<String>, say. Fortunately, Java class files record this information, so no more sophisticated analysis (such as our sizeofness analysis in §5.2) is necessary.

9.3 Combining with memory checks

Integrating libcrunch with Jones & Kelly-style spatial memory checking is an obvious next step. A likely pragmatic approach would be to treat arithmetic or indexing operations like casts when they first occur on a given pointer within a given function. Since libcrunch must necessarily discover the allocation bounds (including subobject bounds) when checking this cast, it can return the bounds to the caller. As new pointers are derived from the initial pointer, their bounds will be easily computed from this inisial bound; this can then be used to perform subsequent checks locally and to optimise away redundant checks using the usual compiler optimisation.

Temporal memory safety is harder, because if done naively it requires invalidation of type and bound information which may be dispersed through the process image. One likely avenue is to employ a partitioning-based scheme (similar to that of SAFECode [**?**], but with a different partitioning criteria) which limits this dispersal by grouping heap objects according to whether their lifetimes may overlap. However, this has at least two down-sides we have so far been avoiding: the requirement of whole-program analysis at compile time, and the requirement of heap partitioning at run time. Conservative garbage collection is another approach but with the usual risk of leaks

9.4 Dynamic language opportunities

What we have built, in libcrunch, is effectively a runtime which can identify what lies on the end of a pointer, and can do so reasonably quickly. It is very much the intention that this be used as a basis for dynamic language implementation. The ideas behind libcrunch arose in the context of DwarfPython [Kell and Irwin 2011], and we plan to rebase the latter onto our infrastructure, with the goal of producing a competitively-performing implementation of Python with much more seamless integration with native code and whole-program debuggers.

9.5 Tool-building opportunities

For the same reasons

9.6 Information hiding and levels of abstraction

What we are not doing is "type structure" in the sense of Reynolds: we are not enforcing levels of abstraction. (This is also "information hiding" in the sense of Parnas. Maybe don't bring Reynolds into it?)

For example, our __is_a check will pass for *any* non-erroneous interpretation of the data residing at the target address, regardless of whether interpreting the memory this way would be an abstraction violation from the perspective of information hiding. We ideally instead want to restrict different callers to different abstract views of a given piece of state. Instead of whether p __is_a X, clients should really ask whether p __is_visibly_a X, i.e. accounting for information hiding constraints. In general this depends not only on the data type, but who instantiated it (and context-sensitively, i.e. on whose behalf). For example, the details of a specific linked list data type might be visible to client Z for its own list of widgets, but the presence of the same linked list data type used from deep inside the C library, say, ought to be hidden from Z.

A special case of this which our system's own internals exposes concerns "meta-levels" of execution [**?**]. In general, instrumentation-based programs ought to be isolated from the programs they observe, but unavoidably share the same underlying execution environment. This makes abstraction violations particularly easy to construct. Our treatment of allocations presents (at least) two ways in which meta-levels are confused. One is how we insert trailers into heap chunks (§6.3), but are unable to ask libcrunch to verify that a pointer indeed points to a trailer. Another is how a nested allocator (A third might be how the char** free-list trick of one of the SPEC benchmarks isn't handled gracefully. A fourth might be how we resort to using mmap() to manage our own dynamic memory, and can't answer queries like __is_a about our own allocations. We should just have a well-founded hierarchy meaning that we index our own mmap'd regions, and it all terminates nicely

)

10. Conclusions

Acknowledgments

REMS. Oxford Martin School. Doug Lea

References

G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

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. ISBN 3-540-00904-3.

D. Eberly. Fast inverse square root (revisited). Web note, 2010. URL http://www.geometrictools.com/Documentation/FastInverseSqrt.pdf. Retrieved on 2014/3/15.

S. I. Feldman. Make: a program for maintaining computer programs. *Softw: Pract. Exper.*, 9, 1979. ISSN 1097-024X. doi: 10.1002/spe.4380090402.

S. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003. ISBN 0521826144.

S. Kell and C. Irwin. Virtual machines should be invisible. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 289–296, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1183-0. doi: 10.1145/2095050.2095099. URL http://doi.acm.org/10.1145/2095050.2095099.

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. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542504. URL http://doi.acm.org/10.1145/1542476.1542504.

V. Saraswat. Java is not type-safe. Web note, 1997.

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

B. Stroustrup. *The C++ programming language*. Addison-Wesley, 1997.

T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 377–388, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi: http://doi.acm.org/10.1145/1706299.1706343.