Posted on July 21, 2014 by Michael Hicks

← Previous Next →

What is memory safety?

I am in the process of putting together a MOOC on software security, which goes live in October. At the moment I'm finishing up material on buffer overflows, format string attacks, and other sorts of vulnerabilities in C. After presenting this material, I plan to step back and say, "What do these errors have in common? They are violations of *memory safety*." Then I'll state the definition of memory safety, say why these vulnerabilities are violations of memory safety, and conversely say why memory safety, e.g., as ensured by languages like Java, prevents them.

No problem, right? Memory safety is a common technical term, so I expected its definition

would be easy to find (or derive). But it's much trickier than I thought.

My goal with this post is to work out a definition of memory safety for C that is semantically clean, rules out code that seems intuitively unsafe, but does not rule out code that seems reasonable. The simpler, and more complete, the definition, the better. My final definition is based on the notion of *defined/undefined memory* and the use of *pointers as capabilities.* If you have better ideas, I'd love to know them!

For the purposes of this post, we are generally considering whether a program *execution* is memory safe or not. From this notion, we deem a program to be memory safe if it all of its possible executions are memory safe, and a language to be memory safe if all possible programs in the language are memory safe.

Not in my house

The recent *Systematization of Knowledge (SoK)* paper, the Eternal War in Memory, exemplifies one common way of defining memory safety. It states that a program execution is memory safe so long as a particular list of bad things, called memory access errors, never occur:

1. buffer overflow
2. null pointer dereference
3. use after free
4. use of uninitialized memory
5. illegal free (of an already-freed pointer, or a non-malloced pointer)

The wikipedia page on memory safety provides a

similar definition. This is a good list, [1] but as a definition it is not very satisfying. Ideally, the fact that these errors are ruled out by memory safety is a *consequence* of its definition, rather than the *substance* of it. What is the idea that unifies these errors?

No accesses to undefined memory

One unifying idea might be the following: a memory error occurs when the program accesses *undefined memory*, which is memory that the program has not specifically allocated, i.e., as part of the heap (through malloc), stack (as a local variable or function parameter), or static data area (as a global variable). George Necula and his students, as part of their work on CCured, a project aiming to enforce memory safety for C programs, thus proposed that a memory safe program execution is one that never accesses undefined memory. We can assume that memory is conceptually infinitely large, and that memory addresses are never reused. As such, memory that is freed (e.g., by calling **free**, or by popping a stack frame when returning from a function) is never reallocated and stays permanently undefined.

This definition clearly rules out errors 2 and 3, and rules out 4 if we extend our definition of “allocated” to include “initialized.” Error 5 is ruled out if you assume that **free** can only be called on pointers to defined memory.

Unfortunately, this definition does not rule out buffer overflows, at least not in practice. For example, assuming a standard stack layout, *Program 1*‘s execution would be considered memory safe by

this definition:

```
1  /* Program 1 */
2  int x;
3  int buf[4];
4  buf[5] = 3; /* overwrites x */
```

The definition allows this program because it is writing to legally allocated memory; it is even writing to it at the correct type. The problem is that it is writing to variable **x** by overflowing **buf**, and intuitively this would seem to be memory unsafe.

Infinite spacing

We can slightly extend the definition to rule out the above program by adding, for the purposes of the definition, the assumption that memory regions are allocated infinitely far apart.

Assuming infinite spacing would render the above program to be memory unsafe. For the purposes of our definition we should view **buf** and **x** as allocated infinitely far apart. As such, the **buf[5]** access is to memory outside of **buf**'s region, and into undefined memory, and thus an error. Overflows of objects allocated on the heap, or static data area, are handled similarly. Note that infinite spacing effectively prevents forging pointers directly from integers because it would be infinitely unlikely that you could pick an integer that corresponds to a legal memory region. Emery Berger and Ben Zorn, in their work on the DieHard memory allocator, aim to enforce this definition of memory safety probabilistically, by randomly spacing allocated objects.

While we are much closer to a satisfying definition, we

are still not there yet. In particular, the definition would seem to allow the buffer-overflowing *Program 2*, which is a variant of *Program 1*, above.

```
1  /* Program 2 */
2  struct foo {
3    int buf[4];
4    int x;
5  };
6  struct foo *pf = malloc(sizeof(stru
7  pf->buf[5] = 3; /* overwrites pf->x
```

Here, the buffer overflow is *within* an object. We could similarly rule out the buffer overflow by appealing to infinite spacing between record fields. This is not so divorced from reality, as the C standard allows the compiler to decide how to place padding between fields. On the other hand, the language suggests that a record is a single object (a single pointer is returned from **malloc**). Many programs will cast one struct to another, or assume a certain padding scheme. Both operations are supported by many compilers, despite the freedom afforded by the standard. Therefore we would prefer a definition that does not rely on an abstraction that is so from reality.

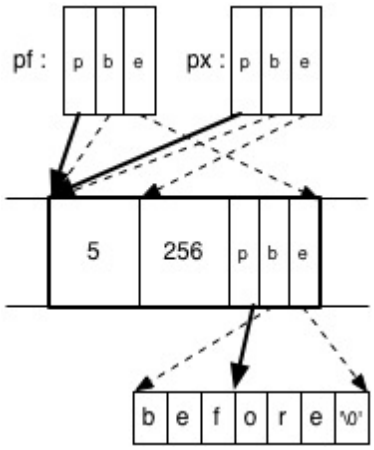
Pointers as capabilities

The definition I prefer, which I'll now present, is inspired by Santosh Nagarakatte's work (done with several collaborators) on SoftBound, an approach for enforcing memory safety in C programs (which has inspired Intel hardware support, coming soon).

As with our first definition, we have a notion of defined (allocated) and undefined (never-allocated or deallocated) memory, where we assume deallocated memory is never reused, and memory safety is

violated if undefined memory is accessed. We add to this a notion of pointers as capabilities; that is, they allow the holder of the pointer to access a certain region of memory. We should think of a pointer as consisting of logically three elements *(p,b,e)*: the legal region is defined by the base *b* and bounds (or *extent*) *e*, and the pointer itself is *p.* The program only truly manipulates *p*, while *b* and *e* are conceptually present for the purposes of defining whether an execution is memory safe.

As an example, here is *Program 3* and a visualization of the (conceptual) memory that results from running it.




```
1  /* Program 3 */
2  struct foo {
3    int x;
4    int y;
5    char *pc;
6  };
7  struct foo *pf = malloc(...);
8  pf->x = 5;
9  pf->y = 256;
10 pf->pc = "before";
11 pf->pc += 3;
12 int *px = &pf->x;
```

The last two lines are most interesting. The program

is allowed to employ pointer arithmetic to create new pointers, but may only dereference them if they stay within the allotted range *b* to *e*. In the example, we incremented **pc**'s *p* component, but it retains the *b* and *e* components it started with; as such, executing ***(pf->pc)** would be legal. If instead we'd done **pf ->pc += 10**, then ***(pf->pc)** would constitute a violation of memory safety even if **pf->pc** happened to point to defined memory (allocated to some other object).

Program 3's last line creates a pointer **px** to the first field of **pf**, narrowing the bounds to just that field. This rules out overflows like that in *Program 2*. If instead we had retained the bounds of **pf** the program could use **px** to overflow into the other fields of the struct.

A capability is not forgeable, and likewise we should not be able to forge pointers by casting them from integers. Illegal casts could be *direct*, e.g., by doing **p = (int *)5**, or *indirect*, e.g., by casting a struct containing integers to an object containing pointers, e..g, by doing **p = (int **)pf** to effectively cast the first field of the struct in *Program 3*, defined as an integer, to be a pointer. Our definition simply treats casts as no-ops; only legal pointers can be dereferenced, and a pointer's capabilities are determined when it is created. Thus our definition permits *Program 4*.

```
1  /* Program 4 */
2  int x;
3  int *p = &x;
4  int y = (int)p;
5  int *q = (int *)y
6  *q = 5;
```

Conceptually, the base and bounds originally given to

p will stay with it even when it is converted into the integer **y**, so that they are present when **y** is converted back to **q** and dereferenced. Dually, if we did **p = (int **)pf** at the end of *Program 3* above, we could follow it with ***p = malloc(sizeof(int))** and then be allowed to do both ****p** and **printf("%d\n",pf->x)**. That is, memory that once contained an integer can be modified to contain a pointer which is then dereferenced, and pointers can be safely treated as integers, but not vice versa.

In a sense, the capability-based definition of memory safety is a form of type safety, where there are only two types: pointer types and non-pointer types. [2] The definition ensures that (1) pointers are only created in a safe way that defines their legal memory region; (2) pointers can only be dereferenced if they point to their allotted memory region; and (3) that region is still defined. And this definition rules out all five error types while allowing for a reasonable set of coding idioms.

What do you think? Does this definition present a good mental model of what one should think of when trying to write a memory-safe C program? Are there important examples of bad code that this definition would consider safe, or examples of the reverse, and if so how should we adjust the definition to accommodate them, while hopefully keeping it simple?

Conclusion

Some of you might be wondering at this point: Has this just been an "academic" exercise? Why is a definition of memory safety useful?

To me, a good definition matters for scientific progress. When I read a technical paper that claims to ensure memory safety, I might think I'm getting one thing when the paper is actually ensuring another. [3] Such confusion impedes progress. For example, when a technical paper develops some technology, e.g., for analyzing C programs, that claims it is correct *assuming* memory safety, we cannot really challenge that claim unless we know what memory safety is. The papers Static Error Detection using Semantic Inconsistency Inference and Large-Scale Analysis of Format String Vulnerabilities in Debian Linux claim to have a *sound* analysis (more on this idea in my previous post on Heartbleed) assuming the underlying program is memory safe. What definition of memory safety are they thinking of?

The thinking contained in this post has benefited from discussions with Emery Berger, Evan Chang, Derek Dreyer, Jeff Foster, Dan Grossman, Milo Martin, Kris Micinski, Andrew Miller, Greg Morrisett, Andrew Myers, Santosh Nagarakatte, James Parker, Aseem Rastogi, John Regehr, Ken Roe, Andrew Ruef, Peter Sewell, Yannis Smaragdakis, Ross Tate, and probably a few others I've forgotten. Thanks, guys!

OK, now back to my course prep …

Notes:

1. The wikipedia page adds out-of-memory errors to its list of memory safety violations. I don't agree with this addition; type-safe languages like Java, which use garbage collection, are universally viewed as memory safe, and yet they have out-of-memory errors too. ↩

2. An important proviso is that function pointers may not be the same as normal pointers. I haven't thought long and hard about it, but it would be reasonable to suggest that a program that casts an **int*** pointer to a function pointer and then calls the function is not memory safe. On the other hand, this might actually work: actions taken by those integers interpreted as instructions are fine as long as they follow the rest of the rules. So perhaps **int*-**to-function-pointer is memory safe, but not type safe. ↩
3. I was surprised to find papers like Implementation of the Memory-safe Full ANSI-C Compiler and Memory safety without garbage collection for embedded applications which use the term memory safety in the title, and yet never carefully define the term in their paper. I don't mean to pick on these two papers; they constitute very good work, and are by no means the only papers not to clearly define what they mean. But they are an example of the problem I am trying to solve here. ↩

This entry was posted in **PL in practice**, **Semantics**, **Software Security** and tagged **Bug-finding**, **C programming language**, **memory safety**, **semantics**, **software security** by **Michael Hicks**. Bookmark the **permalink**.

44 THOUGHTS ON "WHAT IS MEMORY SAFETY?"

J. Ian Johnson on **July 21, 2014 at 8:39 am** said:

Your musing about function pointers is intriguing, but there is a hidden aspect that you open your language model to state that code not only lives in memory (instead of being a special component of the abstract machine), but code is encoded as machine words that throws the abstract machine into a different mode of execution if you start executing these words blessed as “code.” It would take some particular cleverness in the abstract machine semantics to escape this encoded execution mode, since everything is words from then on. You could have a machine representation that pairs your language’s code to a “correct compilation” of it, but that is a huge can of worms if you want to allow whole-program analyses to break abstractions (say, a function isn’t as general as its high-level code because we can determine it will never be used in a certain way). The correct way to model this high-level-interacting-with-low-level behavior is a question I’d like Amal Ahmed or Andrew Appel to weigh in on.

Reply ↓

Michael Hicks
on **July 21, 2014 at 8:43 am** said:

Good call, Ian. I agree that it would be simplest to think of code entirely in a separate address space, and thus designate an attempt to call a function not in that address space to be a violation. Zhong Shao has also thought a lot about self-modifying and code-generating programs. Curious what he thinks.

Reply ↓

Brendan on **July 21, 2014 at 2:32 pm** said:

Does this definition outlaw the linux kernel's ubiquitous lists based on container_of, where the list is a pointer to a field in a struct and you get a pointer to an element by subtracting the offset of the list field from the list pointer?

Reply ↓

Michael Hicks
on **July 21, 2014 at 3:11 pm** said:

I believe it's allowed with the caveat you need to specify the size of the "sub-bounds" precisely. A related idiom is having something like struct foo { int x; int y; } *pf and then wanting to cast &pf->x to px of type int*, where px[0] *and* px[1] should be allowed. By default, you'd only get px[0]. You can imagine the programmer indicating that the "overflow" is allowed (through some other language mechanism), and thus not a violation.

Reply ↓

Peter
on **July 22, 2014 at 5:50 am** said:

You should be able to interface with a list item without exposing other list items. http://nob.cs.ucdavis.edu/bishop/secprog/robust.html

Reply ↓

Brendan on July 21, 2014 at 2:34 pm said:

Never mind, that's just a p,b,e tuple where b < p < e.

Reply ↓

morrisett on July 21, 2014 at 4:19 pm said:

One question is if you built an abstract machine that enforced the property you've sketched here, how many programs would it break? You can imagine more relaxed notions of memory safety that would break fewer programs, but also allow more attacks. Conversely, we could strengthen the property to something more akin to a "defined" ANSI C program, and we'd break more programs, and perhaps stop more attacks. So I'm curious why you think this is the "right" definition.

Reply ↓

morrisett
on July 21, 2014 at 4:23 pm said:

Just to amplify: Does your abstract machine

support taking x = &A[0] (where A is an array of structs), then passing x to something like memcpy? If you preserve the bounds of the enclosing array, then it would. But such a policy would allow an attacker to potentially overwrite adjacent A values, when we only intend a capability to a particular element.

Reply ↓

Michael Hicks
on **July 21, 2014 at 6:56 pm** said:

The issue here is that the language is ambiguous. When you do p = &A[2] you might mean that you want p to be a pointer to just element 2, or you want it to be a pointer to all of the elements, starting at that one. The mechanics of the definition I've proposed can handle either interpretation. You just have to say which one you mean when determining whether there was a violation. If you say you meant just a single element, then p[1] is a violation, whereas if you meant the rest then it may not be.

Reply ↓

Michael Hicks
on **July 21, 2014 at 6:59 pm** said:

> I agree with the tradeoff you've identified. I've been trying to find a definition that accepts as many programs as possible while ruling out the standard errors people point to when saying "that's a memory safety violation." Some reasonable programs might not be memory safe by my definition, and that could be OK. They might be acceptable by some other definition ("weak memory safety"?) that is not otherwise strong enough to rule out problematic programs in general, but rules out obviously-broken ones.
>
> **Reply ↓**

Todd Millstein on **July 21, 2014 at 9:27 pm** said:

I agree that this is a nice mental model for programmers, and I teach a similar kind of "logical model of memory" in my classes. But at the same time, the capability model feels more like a particular discipline for ensuring memory safety, rather than the definition of memory safety itself. Your "no accesses to undefined memory" seems much closer to such a definition to me. We can argue that programs satisfying this definition but not your capability-based definition are bad style, hard to understand, etc. but it's unclear in what sense they are "unsafe."

I think what the capability style does well is define a memory error in a platform-independent way. That's exactly what makes it a good discipline for programmers. In contrast, whether a particular dynamic memory access violates the "no accesses to

undefined memory" property can depend on the compiler used, the machine we are running on, etc. But that doesn't make it a bad definition of memory safety.

Reply ↓

Michael Hicks
on **July 22, 2014 at 6:32 am** said:

Todd, I have found that people coming from the separation logic community have this view. Basically, they (and you, I think) advocate the defined/undefined memory notion of memory safety, which would allow you to construct pointers from integers (and using xor and other tricks) so long as you end up landing on defined memory.

I would like to start calling this "weak memory safety." As you say, it's technically safe (i.e., you won't crash the program by accessing undefined memory) but I also believe it's not strong enough to cover what the term "memory safety" has colloquially come to mean, particularly with respect to ensuring security (i.e., no buffer overflows). I believe most people (though not all) would call a program that allows buffer overflows "unsafe," and it's not clear to me how to ensure the absence of buffer overflows in all the ways we want without using capabilities.

The nice thing is that capabilities can be viewed as simply an addition to this weaker

model, hence my proposal to prefix your definition with “weak”.

Reply ↓

Todd Millstein
on **July 22, 2014 at 10:10 am** said:

Thinking more, what makes me uncomfortable about your definition is that (as I understand it) it’s a property of the program structure rather than (solely) its dynamic behavior. In your Program 1, if you replace buf[5] = 3 with a write to x, the two programs have exactly the same behavior (and indeed have the same compiled code in all likelihood) but you will deem one of the memory accesses as safe and the other unsafe. I’d prefer the notion of safety to be about only the execution behavior of a program. That doesn’t mean languages should allow all of that flexibility, but to me that’s a separate issue.

Reply ↓

Michael Hicks
on **July 23, 2014 at 6:58 am**
said:

Good observation. My definition does indeed rely on program structure. I would

> argue the defined/undefined memory definition does too, but it relies on less: just what constitutes memory allocation and deallocation. But I've already argued why I find that definition unsatisfying, which is that I prefer not to allow an access to legal memory by illegal means, where illegal is defined by … the program structure. This is what a buffer overflow is, and it's the poster-child for a violation of memory safety, in my mind.
>
> **Reply ↓**

David Drysdale on **July 22, 2014 at 3:21 am** said:

On the subject of pointers-as-capabilities, you might find the CHERI hardware platform interesting: http://www.cl.cam.ac.uk/research/security/ctsrd/cheri/

Reply ↓

> 
>
> **Michael Hicks**
> on **July 22, 2014 at 6:53 am** said:
>
> Neat! I see they have implemented the pointers-as-capabilities idea directly with

hardware support. They compare this support to Intel MPX, which should enable similar safety assurances, but I haven't thought hard about the ramifications of the different design choices.

Reply ↓

Neel Krishnaswami on **July 22, 2014 at 5:41 am** said:

Hi Mike,

Take a look at Tuch, Klein and Norrish's POPL'07 paper, "Types, Bytes, and Separation Logic". They give a formulation of separation logic for the (sequential[*]) C memory model, which supports the same limited pointer arithmetic that C specifies (i.e., it's okay, but only if it stays within the bounds of an allocated object). One nice thing about their paper is that it also shows how to use of C types to derive non-aliasing facts when it's safe to do so. You probably don't want to hit your students with separation logic, but it could help you organize your lectures.

[*] AFAIK, there are a number of proposals for the concurrent C memory model, but nothing really satisfactory yet.

Reply ↓

Michael Hicks
on **July 22, 2014 at 6:48 am** said:

> Thanks, Neel! I like this paper's thrust for realism, while retaining tractability for reasoning.
>
> **Reply ↓**

Stephen Kell on **July 23, 2014 at 4:19 am** said:

Hi Mike. I'm really glad you're asking this question, and I think you've done a very nice job with this definition.

I can think of two interesting subtleties. I like the fact that round-tripping a pointer through an integer continues to work, since lots of code does this. So how much computation are we allowed to do on that integer before we give up trying to propagate the bounds info? Can we handle something like the "XOR linked list"? It seems not, at present; should this be accommodated, and if so, how? (I have some ideas and opinions about this, but don't want to prejudice your answer. 😀

Secondly, what is "initialization"? Does it have to remove nondeterminism completely, or just to a well-specified degree? If the latter, we have to worry about information leakage. Is that a memory-safety issue? Recently we had a discussion internally here in Cambridge about the extent to which various languages would prevent Heartbleed (say). It was very clear (to me at least) that talking about "type-safety" was too vague (so I really welcome this post focusing on memory-safety!). But less obviously, even apparently memory-safe languages can generate

Heartbleed-like vulnerabilities. With a massive hat-tip to Thomas Leonard for sharing this observation: OCaml code using String.create can give you a bug very similar to Heartbleed, but leaking information across a heap memory re-use rather than a bounds violation. The reason is that String.create specifies its output string as initially containing “arbitrary characters” (and is implemented by leaking the previous heap contents). Is this an allowable kind of initialization or not?

Another thing is that you could perhaps mention is that your list of what is “defined memory” should not be considered prescriptive, at least if you’re talking about languages like C that are designed not to be a closed box. The authority on what is “defined” within an address space is the operating system and/or hardware, so of course, depending on the environment, various things might be fair game besides what your list mentioned: memory allocated by code in other languages, storage allocated by the program loader (not the program), memory-mapped files, memory-mapped hardware registers, etc..

Finally, about your function pointer case: agreed that if instructions are integers, there’s no problem here. You might *get* a memory-unsafe program, but simply *doing* this should not be considered memory-unsafe, nor type-unsafe. If you read/write ints through the int* it might be a violation of information hiding (depending on which code is doing it), and it might corrupt the invariants of the instruction stream or the containing function (if it writes a wrong int), but I’d contend that type-based checks (either static or dynamic) are never sufficient to guard against these kinds of error.

Reply ↓

Michael Hicks
on **July 23, 2014 at 6:56 am** said:

Hi Stephen. Thanks for the great observations!

For the XOR list, I don't think this works. Suppose I have a list with three elements A, B, and C. In the list implementation you would do link(B) = addr(A)⊕addr(C) to construct the link which element B uses to "point" to both elements A and C. In my definition, link(B) will retain the b and e elements from the pointer to element A. If we are traversing the list backwards from B to A you'll reconstruct the p pointer using fancy arithmetic and it will be in bounds. But if you go the other way, you will construct p pointing to element C, which will not be in bounds with the b and e stored in the link. Bummer. This is disappointing because you are not manufacturing the pointer from thin air, but rather manipulating legitimate pointers. Not sure if there's a way to fix this.

I went back and forth on initialization. The definition right now assumes that no memory is being reused, so the information leak you are worried about can't happen. As such, you could actually allow an access to uninitialized memory as long as that use wasn't treating the memory as a pointer. Any other access is a possible violation of type safety, information hiding, etc. but it would be memory safe, so probably I should allow it.

Reply ↓

Zev Weiss
on **July 24, 2014 at 2:43 am** said:

On the XOR-linked-list matter — so XOR of two pointers (capabilities) would be a non-commutative operation? To stretch the bounds of model a bit, could we instead define an XOR of pointers as yielding a new type (call it a pointer-pair, for lack of anything better) structured as a tuple (p, b1, e1, b2, e2) to preserve both regions of memory that would be valid for access via a plain pointer reconstructed from it? I guess you'd then define XOR between a pointer-pair A and a pointer B (for the pointer-reconstruction) as yielding a pointer C where C's b/e values are whichever pair of A's b/e values don't match B's b/e (or some propagating indeterminate value if neither pair matches).

If you wanted to go further (e.g. to accommodate XORing a pointer-pair with another pointer-pair) I suppose you'd need to correspondingly increase the number of b/e pairs stored alongside each p…

Haven't fully thought this through, but I wonder if you could just generalize to

all pointers being (p, s), where s is a set of b/e pairs, and an XOR of two such pointers A and B would yield a new pointer C where p(C) = p(A) XOR p(B) and s(C) = union(s(A), s(B)) – intersection(s(A), s(B))…

Reply ↓

Michael Hicks
on **July 24, 2014 at 6:32 am**
said:

Interesting idea! I wonder if there are other operations, aside from + and XOR, that you might reasonably do to pointers? I also wonder how often the XOR trick happens in the wild (does it?).

Reply ↓

Zev Weiss
on **July 24, 2014 at 10:27 am** said:

Not sure about real-world use of XORed pointers, but I'd bet there are plenty of dynamic-language implementations out there that use ANDing and ORing to steal a

few low-order bits from known-aligned pointers to store the type of the pointed-to objects in them.

And while it was only for a (somewhat creative interpretation of a) school assignment and not anything “real”, I’ve written code that effectively kind of emulates the x32 style of 32-bit pointers in otherwise 64-bit code by sharing the upper 32 bits of all pointers so as to be able to store them more compactly…

exceptttheweasel on **July 23, 2014 at 11:07 am** said:

Are you familiar with Rust? It’s a systems programming language with memory safety as one of the key design goals: http://www.rust-lang.org/

Reply ↓

Michael Hicks
on **July 24, 2014 at 6:28 am** said:

Yep, and I like what I've seen so far. We are starting to do some research with it now in my group.

Reply ↓

emeryberger on **July 23, 2014 at 5:55 pm** said:

Nice post. Have to say I was tickled by the comment "we would prefer a definition that does not rely on an abstraction that is so from reality." A DieHard-based system (Windows' Fault-Tolerant Heap) is deployed on over 1 billion computers worldwide :).

Reply ↓

Michael Hicks
on **July 24, 2014 at 6:30 am** said:

DieHard is great! Now you have to figure out how to make a version that works for the stack, and between fields in structs, without using too much physical memory.

Reply ↓

emeryberger
on **July 24, 2014 at 10:15 am** said:

We know how to make it work for the stack – we did something like this for Stabilizer (http://dl.acm.org/citation.cfm?id=2451141), a system that enables statistically sound performance evaluation by continuously randomizing damn near everything. But it takes some intimacy with the compiler or binary rewriting. Physical memory footprint is not really a problem (especially these days!) — the only issue (for some programs) is the TLB. But in general, even randomizing nearly everything repeatedly during program execution — is surprisingly cheap.

The above do not alter C/C++ semantics, so they are fine. But if you don't want to break programs, you can't alter layout within structs. Fundamentally, because C and C++ bake memory layout into their type system (e.g., padding, alignment, relative positions of fields), there's no way to fully divorce type safety and memory safety in those languages. {Thanks to Sam Guyer for the pithy last observation.}

Reply ↓

Michael Norrish on **July 23, 2014 at 9:35 pm** said:

Apart from XOR tricks, my favourite piece of C standardese (= madness) at http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm also suggests you can't use a formalism along the lines you want to (though I agree your idea is pretty reasonable); you have to track where the pointers came from statically. All so compilers can do simple data-flow analyses and optimise code as a result.

Reply ↓

Zev Weiss on **July 24, 2014 at 2:09 am** said:

Minor nitpick — I'm guessing the various 'buf[5]' references in Programs 1 & 2 (and surrounding prose) were intended to be 'buf[4]'? (Or alternately, the declarations 'int buf[6];'.)

Reply ↓

Michael Hicks
on **July 24, 2014 at 6:33 am** said:

That was just to make it abundantly clear that we've run off the end (ahem). Anything greater than 3 works, right?

Reply ↓

Zev Weiss
on **July 24, 2014 at 10:29 am** said:

> For a simple overrun, sure — but the "/* overwrites x */" comment appeared to indicate it was intended to overrun by exactly one element.
>
> Reply ↓

Pingback: The Latest News from Coders Weekly, Issue 4Coders Weekly | Coders Weekly

Phuong on July 25, 2014 at 8:33 pm said:

Modern computer need safe memory system. I think best for each memory location to have passcode and the CPU must supply that passcode before it is allowed to write to that location. Computers are fast enough now to provide this without any impact on application software.

Reply ↓

rich on July 26, 2014 at 9:35 am said:

This is essentially the same definition of memory safety as we used in our Bounds Checking GCC back in 1995 (google for: Richard Jones, Paul Kelly), except that the way we implemented it you don't need "fat pointers".

Reply ↓

Gabriel Scherer on **July 30, 2014 at 6:39 am** said:

I think the real question is: what is memory? What is a good memory model? There has been a fair amount of work on various memory models (especially since the advent of the work on mechanized compilers for realistic languages). Given a reasonable memory model, you get a reasonable notion of memory (un)safety: an unsafe access is one that cannot be given a meaning in the model. To answer your question, you in fact defined a memory model.

Reply ↓

Pingback: What is type safety? | The PL Enthusiast

Daniel R. Grayson on **August 8, 2014 at 9:07 am** said:

Here's my favorite definition of memory safety, etc. A programming language is said to be "safe" if the source code of a program, the definition of the programming language, and the data input to the program completely determine the actions of the program (i.e., what the program prints, whether the program crashes, but not how long it takes.).

Reply ↓

Pingback: 型安全性とは何か | POSTD

Pingback: Bridging Algorithms and Programming Languages - The PL Enthusiast

Jarkko Sakkinen on **October 28, 2014 at 3:47 pm** said:

Consider:

#include

int main(void)
{
for (;;) {
char *fc = malloc(512);
fc += 128;
free(fc);
}

exit(0);
}

I believe this is a corner case that is not modeled very well by this definition.

Also, I started to wonder whether memory leaks should be somehow dealed?

Reply ↓

Michael Hicks
on **October 29, 2014 at 7:05 am** said:

I'm not sure what your program is meant to illustrate. Perhaps you are saying there is a problem of freeing a pointer into the middle of an object? (If so, I'm not sure why you need the loop, so maybe I'm missing something.)

I have been vague in this post about how

free() is used in the definition of temporal safety because it depends on the implementation of free(). Some implementations of free() can tell that you've given them a pointer into the middle of an object, and will free the whole object properly. Other implementations will detect the invalid pointer, and crash/fail. Others simply can't tell, and may behave badly.

To keep things simple, imagine that we extend the definition to say that free() must be given a pointer returned by malloc(), and if not, that's an error. I'm not convinced that it makes sense to view it as a memory safety error though (rather than we assume it follows as a precondition of our memory safety definition) because it's an error in the way malloc()/free()'s implementation is used, not the concept of when memory is used safely.

Memory leaks are not properly part of memory safety because leaks do not compromise our ability to reason about the program properly, in terms of enforcing memory boundaries. They are a coding mistake, but they are not "undefined" in the same way that dangling pointer dereferences and buffer overflows are. Another way to look at it is that type-safe languages, which use garabge collection, have memory leaks, too.

Reply ↓

nemobis on **October 30, 2014 at 3:59 pm** said:

Maybe it's me, or maybe I misunderstood everything, but I find the terminology unnecessarily confusing: "base" is ambiguous and "extent" makes me think of norm; why can't you just call them beginning, end? Then $p \in [b, e]$; or better yet, $x \in [a, b]$.

Reply ↓

Rob on **November 1, 2014 at 9:20 pm** said:

I read about pointers today in relation to Java. Some of this is beginning to make sense.

Reply ↓

Leave a Reply

Enter your comment here...