

CHECKEDCBOX: Formalizing RLBox in Checked C for Incremental Spatial Memory Safety (Extended Version)

This is an extended version of a paper that appears at the 2022 Computer Security Foundations Symposium.

1. Introduction

Vulnerabilities due to memory corruption, especially spatial memory corruptions, are still a major issue for C programs [3, 32, 33] despite a large body of work that tries to prevent them [30]. Several industrial and research efforts, including CCured [25], Softbound [23], and ASAN [28], have investigated ways to compile C programs to automatic spatial safety enforcement. These approaches all impose performance overheads deemed too high for deployment use. Recently, Elliott et al. [7] and Li et al. [16] introduced and formalized Checked C, an open-source extension to C with new types and annotations whose use can ensure a program’s spatial safety. Importantly, Checked C supports development that is incremental and compositional. Code regions (e.g., functions or whole files) designated as *checked* enforce spatial safety in a manner preserved by composition with other checked regions. But not all regions must be checked: Checked C’s annotated *checked pointers* are binary-compatible with legacy pointers, and may coexist in the same code, which permits a deliberate (and semi-automated) refactoring process. Parts of the FreeBSD kernel have been successfully migrated to Checked C [6], and the overall performance overhead is minimal enough for practical deployment.

The guarantee Checked C provides focuses on checked code regions, where any stuck (i.e., ill-defined) state reached by a well-typed program amounts to a spatial safety violation. Such a state can always be attributed to, i.e., *blamed on*, the execution of code that is not in a checked core region. However, what are the guarantees that unchecked (unsafe) code regions can provide? Programmers might not expect to convert every code to Checked C, but leave some functions in unsafe regions, in which case the unsafe region executions should not affect other parts. In this paper, we cover this gap by introducing CHECKEDCBOX, combining Checked C with program partitioning mechanism, making three main contributions.

Executions Cannot Crash. *MZ: This paragraph sounds incoherent to me.* Our first contribution is a clear program partition between checked and unchecked code regions ~~and ensures~~ so that a CHECKEDCBOX program execution *cannot crash* due to spatial safety violations.

In CHECKEDCBOX, heap is partitioned into two parts:

a checked heap containing all checked pointers and an unchecked heap that is sandboxed. Then, we restricted pointer usages so that checked pointers can only refer to checked heap region and unchecked ones can only be used in unchecked heap region. To communicate safely between those two isolated regions, we developed tainted pointers that is always verified upon accessing data and program in unchecked region. We restrict the checked and unchecked pointers usage only in checked and unchecked code regions, respectively; and develop tainted pointers for the communication between the two regions. Heap is partitioned into two parts: a checked heap containing all checked pointers and an unchecked heap that is sandboxed. Tainted pointers exist in the unchecked heap region and must be verified each time being used.

Formalizing the Type System, Semantics and Compiler. Our second contribution is a core formalism named CORECHKCBOX, which extends Li et al. [16] with the non-crashing guarantee and other new features below. We prove formally the *non-crashing theorem*, i.e., a well-typed CORECHKCBOX program can never crash due to spatial safety violations. We maintain the compiler formalism with the new features extended in CHECKEDCBOX. Especially, we maintain the bound-check insertions for array accesses without “fat” pointer compilation scheme, while keeping the non-crashing guarantee. To certify the simulation relation between the CORECHKCBOX semantics and the compiler, we utilize the model-based randomized testing. This is done by a conversion tool that converts expressions from CORECHKCBOX into actual Checked C code that can be compiled by the Clang Checked C compiler. We create a program random generator based on the typing rules of CORECHKCBOX and ensure that CORECHKCBOX and Clang Checked C are consistent after conversion, both statically and dynamically. To the best of our knowledge, CORECHKCBOX is the first C(-like) language and compiler formalism with the program partitioning mechanism.

Supporting Checked Pointer Callbacks and No Checked Pointer Exposure. Our third contribution is an added-up feature to support checked (function) pointer callbacks in unchecked code regions. When designing a multi-threaded system, users might want to provide a third party interface that allows third party developers to create new program features, while keeping these programs in unchecked code regions. Moreover, they do want to provide them a (function) pointer pointing to checked data fields.

However, accessing a checked pointer in an unchecked

region violates the program partition principle of CHECKEDCBOX. To resolve the conflict, we develop two mechanisms in CHECKEDCBOX and maintain a stronger *non-exposure* guarantee on of the non-crashing guarantee; that is, no checked pointer addresses can be observed in an unchecked code region. The first mechanism allows nested checked and unchecked code regions. Users can context switch between checked and unchecked code regions by nested using the keywords `checked` `unchecked`. The type system ensures that no checked pointers can be accessed across the context switching. The second one is that a call to a checked pointer in unchecked code regions must be surrounded by a *tainted shell*; i.e., a tainted function pointer that points to a checked region possibly holding checked pointers. In this case, no checked pointer address will be observed in the unchecked code regions. In CHECKEDCBOX, for every checked function, we automatically compile a tainted version by surrounding the function without a tainted shell. **MZ: What does this mean?** To the best of our knowledge, CHECKEDCBOX is the first work of formalizing C function pointers with security guarantee.

We begin with a review of Checked C and introduction of new features in CHECKEDCBOX (Section 2), present our main contributions (Sections 3–5), and conclude with a discussion of related and future work (Sections 6, 7). All code and proof artifacts (both for Coq and Redex) can be found at <https://github.com/plum-umd/checkedc>.

2. Overview and Transcendence

This section describes Checked C and new features CHECKEDCBOX provides.

2.1. Checked C Overview

Checked C development began in 2015 by Microsoft Research, but it was forked in late 2021 and is now actively managed by the Secure Software Development Project (SSDP). Details can be found in a prior overview [7] or the formalism [16].

Checked Pointer Types. Checked C introduces three varieties of *checked pointer*:

- `ptr<T>` types a pointer that is either null or points to a single object of type T .
- `array_ptr<T>` types a pointer that is either null or points to an array of T objects. The array width is defined by a *bounds* expression, discussed below.
- `nt_array_ptr<T>` is like `array_ptr<T>` except that the bounds expression defines the *minimum* array width—additional objects may be available past the upper bound, up to a null terminator.

A bounds expression used with the latter two pointer types has three forms:

- `count(e)` where e defines the array’s length. Thus, if pointer p has bounds `count(n)` then the accessible memory is in the range $[p, p+n]$. Bounds expression e must be side-effect free and may only refer to variables

whose addresses are not taken, or adjacent `struct` fields.

- `byte_count(e)` is like `count`, but expresses arithmetic using bytes, no objects; i.e., `count(e)` used for `array_ptr<T>` is equivalent to `byte_count($e \times \text{sizeof}(T)$)`
- `bounds(e_l, e_h)` where e_l and e_h are pointers that bound the accessible region $[e_l, e_h)$ (the expressions are similarly restricted). Bounds `count(e)` is shorthand for `bounds($p, p + e$)`. This most general form of bounds expression is useful for supporting pointer arithmetic.

Dropping the bounds expression on an `nt_array_ptr` is equivalent to the bounds being `count(0)`.

The Checked C compiler will instrument loads and stores of checked pointers to confirm the pointer is non-null, and the access is within the specified bounds. For pointers p of type `nt_array_ptr<T>`, such a check could spuriously fail if the index is past p ’s specified upper bound, but before the null terminator. To address this problem, Checked C supports *bounds widening*. If p ’s bounds expression is `bounds(e_l, e_h)` a program may read from (but not write to) e_h ; when the compiler notices that a non-null character is read at the upper bound, it will extend that bound to $e_h + 1$.

Spatial Safety and Backward Compatibility. Checked C is backward compatible with legacy C in the sense that all legacy code will type-check and compile. However, only code that appears in *checked code regions*, which we call *checked code*, is spatially safe. Checked regions can be designated at the level of files, functions, or individual code blocks, the first with a `#pragma` and the latter two using the `checked` keyword.¹ Within checked regions, both legacy pointers and certain unsafe idioms (e.g., *variadic* function calls) are disallowed. The code in Fig. ?? satisfies these conditions and will type-check in a checked region.

How should we approach code that has both checked and legacy components? Li et al. [16] proved, for a simple formalization of Checked C, that *checked code cannot be blamed*: Any spatial safety violation is caused by the execution of unchecked code. **MZ: You might want to add that spatial safety violation is still there with the amalgamation of both checked and unchecked code which gave rise to the motivation of CHECKEDCBOX.**

Converting C to Checked C. The safety guarantees of Checked C come with certain restrictions. For instance, as shown below, Checked C programs cannot use address-taken variables in a bounds expression as the bounds relations may not hold because of possible modifications through pointers.

```
...
array_ptr<int> p : count (n) = NULL;
✗..., &n, .
```

Consequently, converting existing C programs to Checked C might require refactoring, e.g., eliminate `&n` from the program above without changing its functionality. This might require considerable effort [6] depending on the program’s complexity. Recently, Machiry et al. developed 3C [19] that tries to automatically convert a program to Checked

1. You can also designate *unchecked* regions within checked ones.

```

1 //in checked region
2
3 int compare_1(nt_array_ptr<char> x: count (0),
4   nt_array_ptr<char> y : count (0)) {
5   int len_x = strlen(x);
6   int len_y = strlen(y);
7   return sum(x,len_x) < sum(y,len_y);
8 }
9 ...
10
11 int stringsort(
12   nt_array_ptr<nt_array_ptr<char>> s : count (n),
13   ptr<(int)(nt_array_ptr<char>,
14   nt_array_ptr<char>>> cmp, int n) {
15   int i, j, gap;
16   int didswap;
17
18   for(gap = n / 2; gap > 0; gap /= 2) {
19     {
20       do {
21         didswap = 0;
22         for(i = 0; i < n - gap; i++)
23           {
24             j = i + gap;
25             if((*cmp)(s[i], s[j]) > 0)
26               {
27                 int len = strlen(s[i]);
28                 nt_array_ptr<char>
29                   tmp : count (len) = s[i];
30                 s[i] = s[j];
31                 s[j] = tmp;
32                 didswap = 1;
33               }
34             }
35           } while(didswap);
36       }
37     }
38
39   return 0;
40 }

```

Figure 1: Checked stringsort Code

C by adding appropriate pointer annotations. However, as described in 3C, completely automated conversion is *infeasible*, and it requires the developer to convert some code regions manually. Although the backward compatibility of Checked C helps a partially annotated program to enjoy spatial memory safety on those regions using only Checked pointers (i.e., checked, or safe regions), *MZ: then what?*.

2.2. CHECKEDCBOX Transcendence

Here, we discuss three key new features in CHECKED-CBOX with examples.

Maintaining Non-crashing. Previously, the main guarantee of Checked C [16] was the blame theorem. The sources of crashing in Checked C are (1) *crashes in unchecked regions* and (2) the misuse of checked pointers in unchecked regions. For example, at Figure 3 line 31, we passed a checked null-terminated array (NT-array) pointer into an

```

1 //in checked region, tainted version
2 int tainted_compare_1(
3   nt_array_tptr<char> x : count (0),
4   nt_array_tptr<char> y : count (0)) {
5   checked (x,y) {
6     int len_x = strlen(x);
7     int len_y = strlen(y);
8     nt_array_ptr<char> tx : count (len_x)
9       = malloc(nt_array<char>, len_x);
10    nt_array_ptr<char> ty : count (len_y)
11      = malloc(nt_array<char>, len_y);
12    safe_memcpy(tx,x,len_x);
13    safe_memcpy(ty,y,len_y);
14    return compare_1(tx,ty);
15  }
16 }
17 ...
18
19 //calling the function turns
20 //an unchecked region to a checked region.
21 int tainted_stringsort(nt_array_tptr
22   <nt_array_tptr<char>> s : count (n),
23   tptr<(int)(nt_array_tptr<char>,
24   nt_array_tptr<char>>> cmp, int n) {
25   checked (s,cmp,n) {
26     int i;
27     nt_array_ptr<nt_array_ptr<char>> p : count (n)
28       = malloc(nt_array<nt_array_ptr<char>>, n);
29     for(i = 0; i < n; i++) {
30       int len = strlen s[i];
31       nt_array_ptr<char> tmp : count (len)
32         = new malloc(nt_array<char>, len);
33       safe_memcpy(tmp,s[i],len);
34       p[i] = tmp;
35     }
36     ptr<(int)(nt_array_ptr<char> : count (0),
37     nt_array_ptr<char> : count (0))>
38     cfun = find_check(cmp);
39
40     return stringsort(p,cfun);
41   }
42 }

```

Figure 2: Tainted stringsort Code

unchecked function `f`. At line 8, depending on the NT-array size, `free(s[10])` might crash. Even if it does not crash, line 38 is doomed because of the `free` call. *MZ: That's not true: Line 38 assigns the address of a constant char array to `p[10]` which could be valid since `p` is not yet freed.* *MZ: I changed a lot in this paragraph. Please compare with the previous commit.* Enlightened by program partitioning mechanism, we address the crashes caused by spatial safety violation by sandboxing the unchecked code regions and utilize the Checked C type system to rule out checked pointers from any unchecked regions. Besides, we incorporate a new form of pointer, *tainted pointer*, into our existing Checked C system to allow the interaction between checked and unchecked code regions. A tainted pointer can be used in both checked and unchecked region; but it can only point to *unchecked* heap memory so that the

```

1 //in unchecked region
2 int f(char ** s, int (*cmp)(char *,char *),
3     int (*sort)(char **, int (*)(char *,char *),
4         int), int n) {
5 ...
6
7     int i = sort(s,cmp,n);
8     free(s[10]);
9 ...
10 }
11
12 int g(int (*cmp)(char *,char *)) {
13 ...
14     int real_addr = derandomize(cmp);
15 ...
16 }
17
18 int main(int n) {
19     nt_array_ptr<nt_array_ptr<char>> p : count(n)
20     = malloc(nt_array<nt_array_ptr<char>>, n);
21
22     nt_array_tptr<nt_array_tptr<char>>
23     tp : count(n) =
24         tmalloc(nt_array<nt_array_tptr<char>>, n);
25 ...
26     safe_memcpy(tp, p, n);
27
28     unchecked {
29         if (BAD) // a flag to call different funs.
30             //input checked pointers
31             f(p, compare_1, stringsort);
32         else
33             //input tainted pointers
34             f(tp, tainted_compare_1,
35                 tainted_stringsort);
36     }
37
38     if (!BAD) safe_memcpy(p, tp, n);
39     p[10] = "crash?";
40
41     unchecked {
42         if (BAD) g(compare_1)
43         else g(tainted_compare_1);
44     }
45
46     return 0;
47 }

```

Figure 3: Tainted Pointer Usage in Calling Unchecked Fun

integrity of checked region would not be compromised by writes issued by unchecked code. Hence to pass a piece of data from a checked region to an unchecked one using a tainted pointer, instead of exposing the address directly, the user would need to copy it from checked region to the unchecked heap and create a tainted pointer to refer to the copy. For example, we copy the checked pointer data to the tainted pointer `tp` at Figure 3 line 26, and input the tainted pointer to the unchecked function at line 34. At line 8, even if statement might crash, since tainted pointers are stored in the sandboxed heap, it can be recovered. At line

37, the use of a tainted pointer in a checked region requires a verification on it. This is managed by inserting additional checks and creating exception handling before the use by the CHECKEDCBOX compiler. *MZ: Might as well state separately that dynamic checks are instrumented upon dereferencing a tainted pointer in checked region.* Thus, the checked pointer `p` is safely used at line 38.

MZ: The example itself is a bit confusing. There are a few simplifications can be made here: (1) introduce a macro: #define nt_ptr nt_array_ptr to make the example less verbose; (2) split untangle BAD and GOOD code into 2 fragments, listed side-by-side. (3) use more descriptive function name instead of f, g

In CHECKEDCBOX, we prove the non-crashing theorem: any well-typed CHECKEDCBOX program can never crash due to spatial safety violations, i.e., spatial safety violation can be caught at runtime.

MZ: “catching a SSV at runtime” may sound more appealing and more of an application? Assume audience here are still general; they could prefer more application over high-level ideas?

Formalism Formalizing Function Pointers. In C, manipulating function pointers is a major way of implementing high order functions as well as accessing data stored in structs. In previous works, Checked C assumed all function calls are called by name to a global map. *MZ: Global map can be confusing.* In CHECKEDCBOX, we extend the existing formalism with `formalize` function pointers and prove that the type soundness CHECKEDCBOX is preserved.

MZ: I’m trying to make it simpler. Let’s rename compare_1 to cmp_sum. I assume the mininum knowledge of C for readers so that they know sum of string is by the ASCII encoding. Figure 1 defines a string sorting algorithm depending on the input function pointer `cmp` as a string comparator that provides the generic order for strings; and, `cmp_sum` is an example of `cmp` function that compares strings by the sum of all characters in each of them. `compare_1` is an example `cmp` function that adds the ASCII numbers of characters in the two strings and compare the results.

In addition, function pointers enable the callback mechanism, i.e., a server sends a function pointer to a client in an unchecked region, and allows the client to access some server resources by calling back the pointer on demand. This is a common usage between a web-browser and untrusted third-party libraries. The function call in Figure 3 line 34 is one such usage. *MZ: It sounds good. But, we might as well avoid using server/client here. They could potentially divert our motivation from memory-oriented to a network related one or even sync-vs-async discussion.*

Simply adding function pointer type into the existing is not enough. Continuing with the example in Figure 1, we could notice that the type of the argument `cmp` in `stringsort` is not exactly the same with the signature of `cmp_sum`: `cmp` has type `ptr<(int)(nt_array_ptr<char>, nt_array_ptr<char>>>` which takes two NT-array pointers with arbitrary size and outputs an integer, but

`cmp_sum ...` (Wait, what’s the exact difference?) We may need to make it clear here. To make function pointers usable, we also define a subtyping relation in CHECKEDCBOX that permits function pointer subsumption. In this example, the type of `cmp_sum` is automatically casted to the type of `cmp`. In general, if function pointer x has type $*(tl \rightarrow t)$, and y has $*(tl' \rightarrow t')$, in order to use x as y , tl' should be a subtype of tl and t subtypes to t' .

~~We also utilize CHECKEDCBOX subtyping relation to permit function pointer static auto-casting. Function pointer type information might contain array pointer bound information, for which it is inconvenient to coincide the defined types for a function implementation and the function pointer type. For example, the `cmp` argument in `stringsort` (Figure 1) has type `ptr<(int)(nt_array_ptr<char>`,
`nt_array_ptr<char>>`
 meaning that the function takes two NT-array pointers with arbitrary size and outputs an integer. The function pointer to `compare_1` has type `ptr<(int)(nt_array_ptr<char> : count (0),
 nt_array_ptr<char> : count (0))>`.~~

~~To use `compare_1` in `stringsort`, the type is auto-east to the `cmp`’s type. In general, if function pointer x has type $*(tl \rightarrow t)$, and y has $*(tl' \rightarrow t')$, in order to use x as y , tl' should be a subtype of tl and t subtypes to t' . MZ: This could make the subtyping sound more natural.~~

Not Exposing Checked Pointer Addresses. The non-crashing guarantee in CHECKEDCBOX bans the checked pointer manipulation in unchecked code regions. Thus, there is no reason to permit checked pointer variable assignments in unchecked regions; especially, this might expose a checked pointer address to untrusted parties. For example, the call to function g at Figure 3 line 41 lives in an unchecked region, and g might use some mechanisms, such as derandomizing ASLR [29], to achieve the checked pointer address. Thus, it enables a third party to access any checked heap and function data by simple pointer arithmetic. MZ: Is it rather a consequence of forbidding checked pointer manipulation?

We prevent any unchecked regions from acknowledging checked pointer variables to avoid the checked pointer address leak. In addition, to facilitate checked function callbacks, the CHECKEDCBOX compiler compiles every checked function with an additional tainted shell function. Users are required to serve unchecked regions with the tainted shell pointer instead of the original checked function pointer. For example, `tainted_compare_1` and `tainted_stringsort` in Figure 2 are the tainted shells of the checked functions `compare_1` and `stringsort`, respectively. In the tainted shells, the parameter types arguments are the tainted counterparts of versions of the corresponding arguments in the original checked functions. Inside the shell body, the compiler we creates checked pointer copies of the tainted arguments, and call the checked functions. MZ: This part is unclear. What are you trying to copy? Copying pointers doesn’t make differences: pointers are just memory address. It’s copying data that really helps.

In addition, once the checked function returns, if the output is a checked pointer, we copy its data to a new tainted pointer and return it so that the tainted caller only observes a tainted pointer that points to data in unchecked heap. This prevents checked pointer address exposure. Figure 3 line 42 is an example of passing a tainted shell pointer as an argument to the function issued from an unchecked region serving the function call living in an unchecked region with a tainted shell pointer argument `tainted_compare_1`. Even if g derandomizes its address (line 14), the shell address is in the sandbox and has no harm, and because calling the shell never leaks exposes any checked pointer information outside of the shell.

Conceptually, the shell is run in a checked region. Essentially, a tainted shell is a safe closure that contains a checked block. Once the closure is called, the system switches into the checked mode is turned to a checked region. For example, Figure 2 line 25 surrounds the tainted shell body with a checked block indicating that we context-switch MZ: potential term abuse; “switch” would suffice from the unchecked to checked code region; thus, the checked function call at Figure 2 line 14 is safe, even if it is called by g in Figure 3, because it lives in the checked region. In the CHECKEDCBOX formalism, we formalize a checked block on top of the existing unchecked regions, and the transition of a tainted shell call creates a checked block containing the shell body. We also make sure that no arguments, as well as the output, in any these tainted shell contains any checked pointers, as well as no output is of a checked type.

MZ: Shell vs. Wrapper? Shell reminds me of bash.

3. Formalization

Liyi: This section might say too much, we might cut more here, and put more items in the compilation and implementation.

MZ: I am wondering if it’s still a good idea to talk about syntax, semantics and types separately? The key of the work is about tainted pointer, functions and blocks instead of the overview of the entire system. Tearing those topics apart into 3 aspects can lead to incoherence.

This section describes the formal model of CHECKEDCBOX, named CORECHKCBOX, making precise its syntax, semantics, and type system. It also develops CORECHKCBOX’s meta-theory, including the type soundness, non-exposure, and non-crashing theorems.

3.1. Syntax

The syntax of CORECHKCBOX is given by the expression-based language presented in Fig. 4. **Liyi:** why can we just say in..

There are two type notions in CORECHKCBOX. Types τ classify word-sized values including integers and pointers, while types ω classify multi-word values such as arrays, null-terminated arrays, functions, and single-word-size values. **Liyi:** Is this a good way? the differences are

Variables: x	Integers: $n ::= \mathbb{Z}$
Context Mode: $m ::= c \mid u$	
Pointer Mode: $\xi ::= m \mid t$	
Bound: $b ::= n \mid x + n$	
	$\beta ::= (b, b)$
Word Type: $\tau ::= \text{int} \mid \text{ptr}^\xi \omega$	
Type Flag: $\kappa ::= nt \mid \cdot$	
Type: $\omega ::= \tau \mid [\beta \tau]_\kappa \mid \forall \bar{x}. \bar{\tau} \rightarrow \tau$	
Expression: $e ::=$	$n : \tau \mid x \mid e + e \mid (\tau)e \mid \langle \tau \rangle e$ $\mid \text{strlen}(x) \mid *e \mid *e = e$ $\mid \text{let } x = e \text{ in } e \mid \text{if } (e) e \text{ else } e$ $\mid \text{malloc}(\xi, \omega) \mid e(\bar{e})$ $\mid \text{unchecked}(\bar{x})\{e\} \mid \text{checked}(\bar{x})\{e\}$

Figure 4: CORECHKCBOX Syntax

$$\begin{array}{c}
m \vdash \text{int} \\
\\
\frac{\xi \wedge m \vdash \tau \quad \xi \leq m}{m \vdash \text{ptr}^\xi [\beta \tau]_\kappa} \\
\\
\frac{\xi \wedge m \vdash \tau \quad \xi \leq m}{m \vdash \text{ptr}^\xi \tau} \quad \frac{\xi \wedge m \vdash \tau \quad \xi \leq m \quad FV(\bar{\tau}) \cup FV(\tau) \subseteq \bar{x}}{m \vdash \text{ptr}^\xi (\forall \bar{x}. \bar{\tau} \rightarrow \tau)} \\
\\
t \wedge c = u \quad \xi \wedge u = u \quad c \wedge m = m \quad m_1 \wedge m_2 = m_2 \wedge m_1 \\
\xi \leq \xi \quad t \leq \xi
\end{array}$$

Figure 5: Well-formedness for Types

mainly for the result types vs type elements. Pointer types ($\text{ptr}^\xi \omega$) include a pointer mode annotation (ξ , the difference between context and pointer modes is introduced shortly below) that is either checked (c), tainted (t), or unchecked (u), and a type (ω) denoting valid values that can be pointed to. **Liya: a little too much? Should we assume people know what pointer type is? Liya: never mention m .** Array types include both the type of elements (τ) and a bound (β) comprised of an upper and lower bound on the size of the array ((b_l, b_h)). Bounds b are limited to integer literals n and expressions $x + n$. Whether an array pointer is null terminated or not is determined by annotation κ , which is nt for null-terminated arrays, and \cdot otherwise (we elide \cdot when writing types). **Liya: should have a better way.**

MZ: Let's zero in on the function pointer and the mode context formalism. We can say something like "bounds and array ptrs have been studied in [...]" already.

CORECHKCBOX function types ($\forall \bar{x}. \bar{\tau} \rightarrow \tau$) reflect its dependent function declarations, where \bar{x} represents a list of int type variables in a dependent function header that bind bound variables appearing in $\bar{\tau}$ and τ . **Liya: need to fix, do not understand.** We have a well-formed requirement for a function type; that is, all variables in $\bar{\tau}$ and τ are bounded by \bar{x} . Here is the corresponding CHECKEDCBOX syntax for

these types:

```

array_ptr< $\tau$ > : count( $n$ )  $\Leftrightarrow$  ptrt [(0,  $n$ )  $\tau$ ]
nt_array_ptr< $\tau$ > : count( $n$ )  $\Leftrightarrow$  ptrc [(0,  $n$ )  $\tau$ ]nt
tptr<(int)>(nt_array_ptr< $\tau$ > : count( $n$ ),
    nt_array_ptr< $\tau$ >> : count( $n$ ))>
 $\Leftrightarrow$  ptrt ( $\forall n$ . ptrt [(0,  $n$ )  $\tau$ ]nt  $\times$  ptrt [(0,  $n$ )  $\tau$ ]nt  $\rightarrow$  int)

```

As a convention we write $\text{ptr}^c [b \tau]$ to mean $\text{ptr}^c [(0, b) \tau]$, so the first two examples above could be rewritten $\text{ptr}^c [n \tau]$ and $\text{ptr}^c [n \tau]_{nt}$, respectively. **Liya: we might just cut nt-array ptrs. it seems to me that nt-array ptrs are not important.**

CORECHKCBOX expressions include literals ($n : \tau$), variables (x), addition ($e_1 + e_2$), static casts ($(\tau)e$), dynamic casts ($\langle \tau \rangle e$)², the `strlen` operation (`strlen(x)`), pointer dereference and assignment ($*e$) and ($*e_1 = e_2$), resp.), let binding (`let $x = e_1$ in e_2`), conditionals (`if (e) e_1 else e_2`), memory allocation (`malloc(ξ, ω)`), function calls (`e(\bar{e})`), unchecked blocks (`unchecked(\bar{x})\{ e \}`), and checked blocks (`checked(\bar{x})\{ e \}`).

Integer literals n are annotated with a type τ which can be either `int`, or $\text{ptr}^\xi \omega$ in the case n is being used as a heap address (this is useful for the semantics); `0:ptrt ω` (for any ξ and ω) represents the null pointer, as usual. The `strlen` expression operates on variables x rather than arbitrary expressions to simplify managing bounds information in the type system; the more general case can be encoded with a `let`. We use a less verbose syntax for dynamic bounds casts; e.g., the following

`dyn_bounds_cast<array_ptr< τ >>(e , count(n))`
becomes `ptrc [$n \tau$] e .`

MZ: I would say that let's just focus on unchecked and checked. We can make a simple example that contains a function pointer with dependent parameters; then, explain in great detail what that means. Instead of rephrasing the meaning of symbols, we might as well direct the reader to the appendix/previous work for tech details on the last paper.

Liya: should we bring out the difference? since readers might not even know the previous paper. Compared to the former Checked C model [16], there are four differences. First, the CHECKEDCBOX type annotations have well-formed restrictions in Figure 5, for maintaining non-exposure. Mainly, in a nested pointer $\text{ptr}^{\xi_1} (\dots \text{ptr}^{\xi_2} \tau \dots)$, $\xi_2 \leq \xi_1$. It is worth noting that pointer modes are a three point partial order (\leq), where t is the infimum, and $\xi \wedge m$ is a special meet operation that projects pointer modes onto context modes, such that t is projected as u . **Liya: not clear.** Second, `malloc(ξ, ω)` includes a mode flag ξ for allocating different pointers in different heaps. We disallow ω to be a function type ($\forall \bar{x}. \bar{\tau} \rightarrow \tau$). Third, the first expression e in a function call (`e(\bar{e})`) represents a function pointer. **Liya: might need to bring out what is the difference of previous ones.** Fourth, checked blocks are added to the system, which permits the nested context-switching between checked (represented by context mode

2. assumed at compile-time and verified at run-time, see Appendix A

$$\begin{aligned}
e &::= \dots \mid \text{ret}(x, n : \tau, e) \\
r &::= e \mid \text{null} \mid \text{bounds} \\
E &::= \square \mid E + e \mid n : \tau + E \mid (\tau)E \mid \langle \tau \rangle E \mid *E \mid *E = e \\
&\quad \mid *n : \tau = E \mid \text{let } x = E \text{ in } e \mid \text{if } (E) e \text{ else } e \\
&\quad \mid E(\bar{e}) \mid n : \tau(\bar{E}) \mid \text{unchecked}(\bar{x})\{E\} \mid \text{checked}(\bar{x})\{E\}
\end{aligned}$$

$$\frac{m = \text{mode}(E) \quad e = E[e'] \quad (\varphi, \mathcal{H}, e') \longrightarrow (\varphi', \mathcal{H}', e'')}{(\varphi, \mathcal{H}, e) \longrightarrow_m (\varphi', \mathcal{H}', E[e''])}$$

$$\frac{u = \text{mode}(E) \quad e = E[e'] \quad \tau = \text{type}(e')}{(\varphi, \mathcal{H}, e) \longrightarrow_u (\varphi, \mathcal{H}, E[0 : \tau])}$$

$$\begin{aligned}
\text{mode}(E) &= \text{mode}'(E, c) \\
\text{mode}'(\square, m) &= m \\
\text{mode}'(\text{unchecked}(\bar{x})\{E\}, m) &= \text{mode}'(E, u) \\
\text{mode}'(\text{checked}(\bar{x})\{E\}, m) &= \text{mode}'(E, c) \\
\text{mode}'(\alpha(E), m) &= \text{mode}'(E, m) \text{ [otherwise]}
\end{aligned}$$

Figure 6: CORECHKCBOX Semantics: Evaluation

c) and unchecked (represented by context mode u) code regions. One example usage of the nested context-switching is the checked function callbacks inside an unsafe region in Figure 2 and 3. To guarantee the non-exposure safety, we extend the checked and unchecked block syntax to be $\text{checked}(\bar{x})\{e\}$ and $\text{unchecked}(\bar{x})\{e\}$: \bar{x} restricts all free variables appearing in e , and they cannot be checked pointers.

CORECHKCBOX aims to be simple enough to work with, but powerful enough to encode realistic CHECKED-CBOX idioms. For example, mutable local variables can be encoded as immutable locals that point to the heap; the use of $\&$ can be simulated with `malloc`; and loops can be encoded as recursive function calls. `structs` are not in Fig. 4 for space reasons, but they are actually in our model, and developed in Appendix F. C-style `unions` have no safe typing in Checked C, so we omit them. **Liya: last sentence can be cut.**

3.2. Semantics

The operational semantics for CORECHKCBOX is defined as a small-step transition relation with the judgment $(\varphi, \mathcal{H}, e) \longrightarrow_m (\varphi', \mathcal{H}', r)$. **Liya: do we need to say judgment here?** Here, φ is a *stack* mapping from variables to values $n : \tau$ and \mathcal{H} is a *heap* that is partitioned into two parts (c and u heaps), each of which maps addresses (integer literals) to values $n : \tau$. A c pointer is mapped to a heap location in the c heap, while a t and u pointer represents a u heap location. We wrote $\mathcal{H}(m, n)$ to retrieve the n -location heap value in the m heap, and $\mathcal{H}(m)[n \mapsto n' : \tau]$ to update location n with the value $n' : \tau$ in the m heap. It is worth noting that CHECKEDCBOX is not a fat-pointer system; thus, in every heap update, the value type annotation remains the same through program executions. Additionally, for both stack and heap, we ensure $FV(\tau) = \emptyset$ for all the value type annotations τ .

While heap bindings can change, stack bindings are immutable—once variable x is bound to $n : \tau$ in φ , that binding will not be updated. We can model mutable stack variables as pointers into the mutable heap. **Liya: why we need this sentence?** As mentioned, value $0 : \tau$ represents a null pointer when τ is a pointer type. Correspondingly, $\mathcal{H}(m, 0)$ should always be undefined. The relation steps to a *result* r , which is either an expression or a null or bounds failure, represent a null-pointer dereference or out-of-bounds access, respectively. Such failures are a *good* outcome; stuck states (non-value expressions that cannot transition to a result r) characterizing undefined behavior. The context mode m (in \longrightarrow_m) indicates whether the stepped redex within e was in a checked (c) or unchecked (u) region.

The rules for the main operational semantics judgment—*evaluation*—are given at the middle of Fig. 6. The first rule takes an expression e , decomposes it into an *evaluation context* E and a sub-expression e' (such that replacing the hole \square in E with e' would yield e), and then evaluates e' according to the *computation* relation $(\varphi, \mathcal{H}, e') \longrightarrow (\varphi, \mathcal{H}, e'')$, whose rules are given in Fig. 7, discussed shortly. The second rule describes the exception handling for possible crashing behaviors in unchecked regions. A u mode operation can non-deterministically crash and the CHECKEDCBOX sandbox mechanism recovers the program to a safe point ($0 : \tau$) and continues with the existing program state. **Liya: is non-deterministically crashed or non-deterministically transitioned to some place.** Evaluation contexts E define a standard left-to-right evaluation order. (We explain the $\text{ret}(x, \mu, e)$ syntax shortly.) There are other rules for describing the halts of evaluation to null and bounds states in Appendix A. **Liya: might not need this.**

The *mode* function at the bottom of Fig. 6 describes the context mode determination in each evaluation step based on the context E . **Liya: understood?** For any program execution, the function starts the mode computation with c ($\text{mode}(E) = \text{mode}'(E, c)$). The result context mode depends on where \square locates. If it occurs within E in $(\text{unchecked}(\bar{x})\{E\})$ that has no surrounding checked block, the mode is u; otherwise, the mode is c. $\text{mode}'(\alpha(E), m) = \text{mode}'(E, m)$ represent other construct cases that are not checked and unchecked; in such case, the function recursively traverses the sub-context to find the context mode. **Liya: may be too low-level detailed. describe it in a high level.**

MZ: I don't see the thesis of these first a few paragraphs. And, what's the motivation of mode? How on earth this setup solves the problem? If we're explaining reduction in context, shall we delay the introduction of this context mode now?

Fig. 7 shows selected rules for the computation relation. **Checked and Tainted Pointer Operations.** The rules for pointer related operations—S-DEFC, S-DEFT, S-ASSIGNARRC, S-ASSIGNARRT, S-DEFNULL, and S-CAST. The first five define the semantics of deference and assignment operations. Rule S-DEFNULL transitions attempted null-pointer dereferences to null, whereas S-DEFC dereferences a c-mode non-null (single) pointer. When null

$\text{S-DEFC} \quad \frac{\mathcal{H}(\mathbf{c}, n) = n_a : \tau_a}{(\varphi, \mathcal{H}, * n : \mathbf{ptr}^c \tau) \longrightarrow (\varphi, \mathcal{H}, n_a : \tau)}$	$\text{S-ASSIGNARRC} \quad \frac{\mathcal{H}(\mathbf{c}, n) = n_a : \tau_a \quad 0 \in [n_l, n_h]}{(\varphi, \mathcal{H}, * n : \mathbf{ptr}^c [(n_l, n_h) \tau]_{\kappa} = n_1 : \tau_1) \longrightarrow (\varphi, \mathcal{H}(\mathbf{c})[n \mapsto n_1 : \tau_a], n_1 : \tau)}$	
$\text{S-DEFT} \quad \frac{\mathcal{H}(\mathbf{u}, n) = n_a : \tau_a \quad \emptyset; \mathcal{H}; \emptyset \vdash_{\mathbf{u}} n_a : \tau}{(\varphi, \mathcal{H}, * n : \mathbf{ptr}^t \tau) \longrightarrow (\varphi, \mathcal{H}, n_a : \tau)}$	$\text{S-ASSIGNARRT} \quad \frac{\mathcal{H}(\mathbf{u}, n) = n_a : \tau_a \quad 0 \in [n_l, n_h] \quad \emptyset; \mathcal{H}; \emptyset \vdash_{\mathbf{u}} n_1 : \tau}{(\varphi, \mathcal{H}, * n : \mathbf{ptr}^t [(n_l, n_h) \tau]_{\kappa} = n_1 : \tau_1) \longrightarrow (\varphi, \mathcal{H}(\mathbf{u})[n \mapsto n_1 : \tau_a], n_1 : \tau)}$	
$\text{S-DEFNULL} \quad (\varphi, \mathcal{H}, * 0 : \mathbf{ptr}^c \omega) \longrightarrow (\varphi, \mathcal{H}, \mathbf{null})$	$\text{S-CAST} \quad (\varphi, \mathcal{H}, (\tau)n : \tau') \longrightarrow (\varphi, \mathcal{H}, n : \varphi(\tau))$	$\text{S-RETEEND} \quad (\varphi, \mathcal{H}, \mathbf{ret}(x, n : \tau, n' : \tau')) \longrightarrow (\varphi, \mathcal{H}, n' : \tau')$
$\text{S-LET} \quad (\varphi, \mathcal{H}, \mathbf{let} \ x = n : \tau \ \mathbf{in} \ e) \longrightarrow (\varphi, \mathcal{H}, \mathbf{ret}(x, n : \tau, e))$	$\text{S-RETCON} \quad \frac{(\varphi[x \mapsto n : \tau], \mathcal{H}, e) \longrightarrow (\varphi', \mathcal{H}', e')}{(\varphi, \mathcal{H}, \mathbf{ret}(x, n : \tau, e)) \longrightarrow (\varphi'[x \mapsto \varphi(x)], \mathcal{H}', \mathbf{ret}(x, \varphi'(x), e'))}$	
$\text{S-UNCHECKED} \quad (\varphi, \mathcal{H}, \mathbf{unchecked}(\bar{x})\{n : \tau\}) \longrightarrow (\varphi, \mathcal{H}, n : \tau)$	$\text{S-FUNC} \quad \frac{\Xi(\mathbf{c}, n) = \tau(\bar{x} : \bar{\tau})(\mathbf{c}, e)}{(\varphi, \mathcal{H}, n : (\mathbf{ptr}^c \tau)(\bar{n}_a : \bar{\tau}_a)) \longrightarrow (\varphi, \mathcal{H}, \mathbf{let} \ \bar{x} = \bar{n} : (\bar{\tau}[\bar{n}/\bar{x}]) \ \mathbf{in} \ (\tau[\bar{n}/\bar{x}])e)}$	
$\text{S-CHECKED} \quad (\varphi, \mathcal{H}, \mathbf{checked}(\bar{x})\{n : \tau\}) \longrightarrow (\varphi, \mathcal{H}, n : \tau)$	$\text{S-FUNT} \quad \frac{\Xi(\mathbf{u}, n) = \tau(\bar{x} : \bar{\tau})(\mathbf{t}, e) \quad \emptyset; \mathcal{H}; \emptyset \vdash_{\mathbf{u}} n : \mathbf{ptr}^t \tau}{(\varphi, \mathcal{H}, n : (\mathbf{ptr}^t \tau)(\bar{n}_a : \bar{\tau}_a)) \longrightarrow (\varphi, \mathcal{H}, \mathbf{let} \ \bar{x} = \bar{n} : (\bar{\tau}[\bar{n}/\bar{x}]) \ \mathbf{in} \ (\tau[\bar{n}/\bar{x}])e)}$	

Figure 7: CORECHKCBOX Semantics: Computation (Selected Rules)

```

1  nt_array_ptr<char> safe_strcat
2      (nt_array_ptr<char> dst : count(n),
3       nt_array_ptr<char> src : count(0), int n) {
4      int x = strlen(dst);
5      int y = strlen(src);
6      nt_array_ptr<char> c : count(n) =
7          dyn_bounds_cast
8              <nt_array_ptr<char>>(dst, count(n));
9      // sets c == dst with bound n (not x)
10     if (x+y < n) {
11         for (int i = 0; i < y; ++i)
12             *(c+x+i) = *(src+i);
13         *(c+x+y) = '\0';
14         return dst;
15     }
16     return null;
17 }

```

Figure 8: Implementation of safe `strcat`

is returned by the computation relation, the evaluation relation halts the entire evaluation with null (using a rule not shown in Fig. 6); it does likewise when bounds is returned (see Appendix C). S-ASSIGNARRC assigns to an array as long as 0 (the point of dereference) is within the bounds designated by the pointer's annotation and strictly less than the upper bound.

S-DEFT and S-ASSIGNARRT are similar rules to S-DEFC and S-ASSIGNARRC for tainted pointers. Any dynamic heap access of a tainted pointer requires a *verification*. Performing such a verification equates to performing a literal type check for a pointer constant in Figure 11. We explain this shortly in Section 3.3. For now, the verification step,

e.g. $\emptyset; \mathcal{H}; \emptyset \vdash_u n_a : \tau$ in S-DEFC, refers to that the value n_a is well-defined in $\mathcal{H}(m, n_a)$ and has type τ , if τ is a pointer.

Static casts of a literal $n : \tau'$ to a type τ are handled by S-CAST. In a type-correct program, such casts are confirmed safe by the type system no matter if the target is a t or c pointer. To evaluate a cast, the rule updates the type annotation on n . Before doing so, it must “evaluate” any variables that occur in τ according to their bindings in φ . For example, if τ was $\text{ptr}^c [(0, x+3) \text{int}]$, then $\varphi(\tau)$ would produce $\text{ptr}^c [(0, 5) \text{int}]$ if $\varphi(x) = 2$. The full formalism, including struct and null-terminated bound widening pointer operations, is given in Appendix A.

Unchecked and Checked Blocks. Semantically, unchecked and checked blocks act as classical C blocks as rules S-UNCHECKED and S-CHECKED in Figure 7.

Liyi: Yeah, the block semantics is easy, but is there anything to say? I mean this is the essentially item. **MZ:** Explain how context mode works here.

Binding and Function Calls. The semantics manages variable scopes using the special `ret` form. S-LET evaluates to a configuration whose expression is `ret(x, n : τ , e)`. We keep φ unchanged and remember x and its new value $n : \tau$ in e 's scope that is defined by the `ret` operation. **Liyi: understood?** Every time when evaluation proceeds on e (rule S-RETCON), we install the stack value $n : \tau$ for x in φ for the current scope. After one-step evaluation is completed, we store x 's change in the result `ret operation` `ret(x, $\varphi'(x)$, e')`, and restore x 's outer scope value $\varphi(x)$ in φ' . This procedure continues until e' becomes a literal $n : \tau$, in which case S-RETEEND removes the `ret` frame and returns the literal.

Function calls are handled by S-FUNC and S-FUNT, for c and t mode function pointers, respectively. A call to a function pointer n retrieves the function definition in n 's location in the global function store Ξ ,

MZ: *The “global function store” is what’s missing in the background section.*

which maps function pointers to function data τ ($\bar{x} : \bar{\tau}$) (ξ, e), where τ is the return type, $(\bar{x} : \bar{\tau})$ is the parameter list of variables and their types, ξ determines the mode of the function, and e is the function body. **MZ:** *“Function data” is “function definition” right? This sentence is verbose and contains repetitious words?*

Similar to \mathcal{H} , the global function store Ξ is also partitioned into two parts (c and u stores), each of which maps addresses (integer literals) to the function data described above.

The CHECKEDCBX functions are dependent functions. Recall that array bounds in types may refer to in-scope variables; e.g., parameter `dst`'s bound `count(n)` refers to parameter `n` on lines 2-3 in Figure 8. Semantically, the call is expanded into a `let` which binds parameter variables \bar{x} to the actual arguments \bar{n} , but annotated with the parameter types $\bar{\tau}$ (this will be safe for type-correct programs). The function body e is wrapped in a static cast $(\tau[\bar{n}/\bar{x}])$ which is the function's return type but with any integer parameter variables \bar{x} appearing in that type, as type bound variables, substituted with the call's actual arguments \bar{n} . To see why this is needed, suppose that `safe_strcat` in Fig. 8 is defined to return a `nt_array_ptr<int>:count(n)` typed term, and assume that we perform a `safe_strcat` function call as `x=safe_strcat(a,b,10)`. After the evaluation of `safe_strcat`, the function returns a value with type `nt_array_ptr<int>:count(10)` because we substitute bound variable `n` in the defined return type with 10 from the function call's argument list. **Liyi:** *This has appeared in the previous paper. A better description or more items?*

Note that the S-FUNC and S-FUNT rules replace the annotations $\bar{\tau}_a$ with $\bar{\tau}$ (after instantiation) from the function's signature. Using $\bar{\tau}_a$ when executing the body of the function has no impact on the soundness of CORECHKCBX, but will violate Theorem 6, which we introduce in Sec. 4. **Liyi:** *might be cut because no more description on this.* Rule S-FUNT defines the tainted version of function call semantics. In such case, the verification process $\emptyset; \mathcal{H}; \emptyset \vdash_u n : \text{ptr}^c \tau$ makes sure that the function in the global store is well-defined and has the right type.

3.3. Typing

We now turn to the CORECHKCBX type system. The typing judgment has the form $\Gamma; \Theta \vdash_m e : \tau$, which states that in a type environment Γ (mapping variables to their types) and a predicate environment Θ (mapping integer-typed variables to Boolean predicates), expression e will have type τ if evaluated in context mode m . Key rules for this judgment are given in Fig. 9. All remaining rules are given in Appendix B and E.

Pointer Access. T-DEF and T-ASSIGNARR rules examine array dereference and assignment operations respectively, returning the type of pointed-to objects. Rules for pointers for other object types are similar. The condition $m \leq m'$ ensures that checked and unchecked pointers can only be dereferenced in checked and unchecked regions, respectively; The type rules do not attempt to reason whether the access is in bounds; such check is deferred to the semantics.

MZ: *What about tainted pointers?*

Type Equality and Subtyping and Casting. In CORECHKCBX, type equality $\tau =_{\Theta} \tau'$ is a type construct equivalent relation defined by the bound equality ($=_{\Theta}$) in (NT)-array pointer types and the alpha equivalence of two function types in Figure 10. **MZ:** *What’s “type construct equivalent relation”?* Two (NT)-array pointer types $[\beta \tau]_{\kappa}$ and $[\beta' \tau']_{\kappa}$ are equivalent, if $\beta =_{\Theta} \beta'$ and $\tau =_{\Theta} \tau'$; two function types $\forall \bar{x}. \bar{\tau} \rightarrow \tau$ and $\forall \bar{y}. \bar{\tau}' \rightarrow \tau'$ are equivalent, if we can find a same length (as \bar{x} and \bar{y}) variable list \bar{z} that is substituted for \bar{x} and \bar{y} in $\bar{\tau} \rightarrow \tau$ and $\bar{\tau}' \rightarrow \tau'$, resp., and the substitution results are equal.

The T-CASTPTR rule permits casting from an expression of type τ' to a checked pointer when $\tau' \sqsubseteq \text{ptr}^c \tau$. This subtyping relation \sqsubseteq is given in Fig. 10 and is built on the type equality ($\tau =_{\Theta} \tau' \Rightarrow \tau \sqsubseteq_{\Theta} \tau'$). The many rules ensure the relation is transitive. **MZ:** *?* Most of the rules manage casting between array pointer types. The second rule $0 \leq b_l \wedge b_h \leq 1 \Rightarrow \text{ptr}^m \tau \sqsubseteq \text{ptr}^m [(b_l, b_h) \tau]$ permits treating a singleton pointer as an array pointer with $b_h \leq 1$ and $0 \leq b_l$. Two function pointer types are subtyped ($\text{ptr}^c \forall \bar{x}. \bar{\tau} \rightarrow \tau \sqsubseteq_{\Theta} \text{ptr}^c \forall \bar{x}. \bar{\tau}' \rightarrow \tau'$), if the output type are subtyped ($\tau \sqsubseteq_{\Theta} \tau'$) and the argument types are reversely subtyped ($\bar{\tau}' \sqsubseteq_{\Theta} \bar{\tau}$). There is another casting rule in Appendix A stating that users are free to cast types in unchecked code regions, since unchecked regions can contain C code. **Liyi:** *why?*

MZ: *What changes are maded to bound equality compared to CheckedC?*

Since bounds expressions may contain variables, determining assumptions like $b_l \leq b'_l$ requires reasoning about the probable values of these variables'. The type system uses Θ to make such reasoning more precise. **Liyi:** *why?* Θ is a map from variables x to equation predicates P , which have the form $P ::= \text{ge_0} \mid \text{eq } b$. It maps variables to equations that are recorded along the type checking procedure. If Θ maps x to `ge_0`, that means that $x \geq 0$; `eq b` means that x is equivalent to the bound value b in the current context, such as in the type judgment for e_2 in Rule T-LETINT and T-RETINT. Appendix D. has an example rule for populating Θ with a `ge_0` predicate.

Constant Validity. Rules T-CONSTU and T-CONSTC describes type assumptions for constants appearing in a program. $c(\tau)$ judges that a constant pointer in an unchecked region cannot be of a checked type, which represents an assumption that programmers cannot guess a checked pointer address and utilize it in an unchecked region in CHECKEDCBX. **MZ:** *Or simply a checked pointer is not permitted in an unchecked region?* In rule T-CONSTC, we requires a static verification procedure for validating a

T-CONSTU $\frac{\neg c(\tau)}{\Gamma; \Theta \vdash_u n : \tau : \tau}$	T-CONSTC $\frac{\Theta; \mathcal{H}; \emptyset \vdash_c n : \tau}{\Gamma; \Theta \vdash_c n : \tau : \tau}$	T-DEF $\frac{\xi \leq m}{\Gamma; \Theta \vdash_m e : \mathbf{ptr}^\xi \tau}$	T-ASSIGNARR $\frac{\Gamma; \Theta \vdash_m e_1 : \mathbf{ptr}^\xi [\beta \tau]_\kappa \quad \Gamma; \Theta \vdash_m e_2 : \tau' \quad \tau' \sqsubseteq_\Theta \tau \quad \xi \leq m}{\Gamma; \Theta \vdash_m * e_1 = e_2 : \tau}$	T-CASTPTR $\frac{\Gamma; \Theta \vdash_m e : \tau' \quad \tau' \sqsubseteq_\Theta \mathbf{ptr}^\xi \tau}{\Gamma; \Theta \vdash_m (\mathbf{ptr}^\xi \tau) e : \mathbf{ptr}^\xi \tau}$
T-LET $\frac{x \notin FV(\tau') \quad \Gamma; \Theta \vdash_m e_1 : \tau \quad \Gamma[x \mapsto \tau]; \Theta \vdash_m e_2 : \tau'}{\Gamma; \Theta \vdash_m \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau'}$	T-LETINT $\frac{x \in FV(\tau') \Rightarrow e_1 \in \mathbf{Bound} \quad \Gamma; \Theta \vdash_m e_1 : \mathbf{int} \quad \Gamma[x \mapsto \mathbf{int}]; \Theta[x \mapsto \mathbf{eq} \ e_1] \vdash_m e_2 : \tau'}{\Gamma; \Theta \vdash_m \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau'[e_1/x]}$	T-RETINT $\frac{\Gamma[x \mapsto \mathbf{int}]; \Theta[x \mapsto \mathbf{eq} \ n] \vdash_m e : \tau}{\Gamma; \Theta \vdash_m \mathbf{ret}(x, n : \mathbf{int}, e) : \tau}$		
T-CHECKED $\frac{\forall x \in \bar{x}. \neg c(\Gamma(x)) \quad \neg c(\tau) \quad FV(e) \in \bar{x} \quad \Gamma; \Theta \vdash_c e : \tau}{\Gamma; \Theta \vdash_m \mathbf{checked}(\bar{x})\{e\} : \tau}$	T-UNCHECKED $\frac{\forall x \in \bar{x}. \neg c(\Gamma(x)) \quad \neg c(\tau) \quad FV(e) \in \bar{x} \quad \Gamma; \Theta \vdash_u e : \tau}{\Gamma; \Theta \vdash_m \mathbf{unchecked}(\bar{x})\{e\} : \tau}$	T-FUN $\frac{\Gamma; \Theta \vdash_m e : \mathbf{ptr}^\xi \forall \bar{x}. \bar{\tau} \rightarrow \tau \quad \Gamma; \Theta \vdash_m \bar{e} : \bar{\tau}' \quad \bar{e}' = \{e' \mid (e', \mathbf{int}) \in (\bar{e} : \bar{\tau}')\} \quad \forall e'. e' \in \bar{e}' \Rightarrow e' \in \mathbf{Bound} \quad \bar{\tau}' \sqsubseteq_\Theta \bar{\tau}[\bar{e}'/\bar{x}]}{\Gamma; \Theta \vdash_m e(\bar{e}) : \tau[\bar{e}'/\bar{x}]}$		
$c(\mathbf{int}) = \mathbf{false} \quad c(\mathbf{ptr}^c \omega) = \mathbf{true} \quad c(\mathbf{ptr}^\xi \omega) = \mathbf{false} \text{ [otherwise]}$				

Figure 9: Selected type rules

Bound Inequality and Equality:

$$\begin{aligned}
n \leq n' &\Rightarrow n \leq_\Theta n' \\
n \leq n' &\Rightarrow x + n \leq_\Theta x + n' \\
n \leq n' \wedge \Theta(x) = \text{eq } b \wedge b + n \leq_\Theta b' &\Rightarrow x + n \leq_\Theta b' \\
\Theta(x) = \text{eq } b \wedge b' \leq_\Theta b + n \Rightarrow b' &\leq_\Theta x + n \\
b \leq_\Theta b' \wedge b' \leq_\Theta b \Rightarrow b &=_\Theta b'
\end{aligned}$$

Type Equility:

$$\begin{aligned}
&\text{int} =_\Theta \text{int} \\
\omega =_\Theta \omega' &\Rightarrow \text{ptr}^\xi \omega =_\Theta \text{ptr}^\xi \omega' \\
\beta =_\Theta \beta' \wedge \tau =_\Theta \tau' &\Rightarrow [\beta \tau]_\kappa =_\Theta [\beta' \tau']_\kappa \\
\text{cond}(\bar{x}, \bar{\tau} \rightarrow \tau, \bar{y}, \bar{\tau}' \rightarrow \tau') &\Rightarrow \forall \bar{x}. \bar{\tau} \rightarrow \tau =_\Theta \forall \bar{y}. \bar{\tau}' \rightarrow \tau'
\end{aligned}$$

Subtype:

$$\begin{aligned}
\tau =_\Theta \tau' &\Rightarrow \tau \sqsubseteq_\Theta \tau' \\
0 \leq_\Theta b_l \wedge b_h \leq_\Theta 1 &\Rightarrow \text{ptr}^m \tau \sqsubseteq_\Theta \text{ptr}^m [(b_l, b_h) \tau] \\
b_l \leq_\Theta 0 \wedge 1 \leq_\Theta b_h &\Rightarrow \text{ptr}^m [(b_l, b_h) \tau] \sqsubseteq_\Theta \text{ptr}^m \tau \\
b_l \leq_\Theta 0 \wedge 1 \leq_\Theta b_h &\Rightarrow \text{ptr}^m [(b_l, b_h) \tau]_{nt} \sqsubseteq_\Theta \text{ptr}^m \tau \\
b_l \leq_\Theta b'_l \wedge b'_h \leq_\Theta b_h &\Rightarrow \text{ptr}^m [(b_l, b_h) \tau]_{nt} \sqsubseteq_\Theta \text{ptr}^m [(b'_l, b'_h) \tau] \\
b_l \leq_\Theta b'_l \wedge b'_h \leq_\Theta b_h &\Rightarrow \text{ptr}^m [(b_l, b_h) \tau]_\kappa \sqsubseteq_\Theta \text{ptr}^m [(b'_l, b'_h) \tau]_\kappa \\
\bar{\tau}' \sqsubseteq_\Theta \bar{\tau} \wedge \tau \sqsubseteq_\Theta \tau' &\Rightarrow \text{ptr}^\xi \forall \bar{x}. \bar{\tau} \rightarrow \tau \sqsubseteq_\Theta \text{ptr}^\xi \forall \bar{x}. \bar{\tau}' \rightarrow \tau' \\
n' + n = \text{add}(n', n) &\quad (x + n') + n = x + \text{add}(n', n) \\
\text{cond}(\bar{x}, \tau, \bar{y}, \tau') = \exists \bar{z}. \bar{x} \cup \bar{z} \wedge \bar{y} \cup \bar{z} \wedge \text{size}(\bar{x}) = \text{size}(\bar{y}) = \text{size}(\bar{z}) &\quad \wedge \tau[\bar{z}/\bar{x}] = \tau'[\bar{z}/\bar{y}]
\end{aligned}$$

Figure 10: Type Equality and Subtyping

constant pointer, which is similar to the dynamic verification process in Section 3.2. **Liyi: what dynamic verification process?**

The verification process $\Theta; \mathcal{H}; \sigma \vdash_m n : \tau$ checks (Figure 11) validate the constant $n : \tau$, where $\mathcal{H}(m)$ is the initial heap that the constant resides on and σ is a set of constant assumed to be checked. A global function store

$$\begin{aligned}
&\Theta; \mathcal{H}; \sigma \vdash_m n : \text{int} \quad \Theta; \mathcal{H}; \sigma \vdash_m 0 : \text{ptr}^\xi \omega \\
&\frac{(m = c \Rightarrow \xi \neq c) \quad (m = u \Rightarrow \xi = u)}{\Theta; \mathcal{H}; \sigma \vdash_c n : \text{ptr}^\xi \omega} \quad \frac{(n : \text{ptr}^\xi \omega) \in \sigma}{\Theta; \mathcal{H}; \sigma \vdash_m n : \text{ptr}^\xi \omega} \\
&\frac{\text{ptr}^{\xi'} \omega' \sqsubseteq_\Theta \text{ptr}^\xi \omega \quad \Theta; \mathcal{H}; \sigma \vdash_m n : \text{ptr}^{\xi'} \omega'}{\Theta; \mathcal{H}; \sigma \vdash_m n : \text{ptr}^\xi \omega} \\
&\frac{\xi \leq m \quad \Xi(m, n) = \tau(\bar{x}' : \bar{\tau}) (\xi, e) \quad \bar{x} = \{x | (x : \text{int}) \in (\bar{x}' : \bar{\tau})\}}{\Theta; \mathcal{H}; \sigma \vdash_m n : \text{ptr}^\xi (\forall \bar{x}. \bar{\tau} \rightarrow \tau)} \\
&\frac{\neg \text{fun_t}(\omega) \quad \xi \leq m \quad \forall i \in [0, \text{size}(\omega)) . \Theta; \mathcal{H}; (\sigma \cup \{(n : \text{ptr}^\xi \omega)\}) \vdash_m \mathcal{H}(m, n + i)}{\Theta; \mathcal{H}; \sigma \vdash_m n : \text{ptr}^\xi \omega} \\
&\text{fun_t}(\forall \bar{x}. \bar{\tau} \rightarrow \tau) = \text{true} \quad \text{fun_t}(\omega) = \text{false} \text{ [otherwise]}
\end{aligned}$$

Figure 11: Verification/Type Rules for Constants

$\Xi(m)$ is also required to check the validity of a function pointer. **Liyi: Do we discuss this $\Xi(m)$ before?** A valid function pointer should appear in the right store region (c or u) and the address stores a function with the right type. **Liyi: how?** The last rule in Figure 11 describes the validity check for a non-function pointer, where every element in the pointer range ($[0, \text{size}(\omega))$) should be well typed. **MZ: Name those judgements, instead of using “last rule”. If $\mathcal{H}(m, n + i)$ refers to the location on m mode heap, I would prefer \mathcal{H}_m . Liyi: only well typed? Can we have a better way to say this? MZ: Is there a typo in the type rule? Is that supposed to be $n : \tau$ rather than $n : \tau : \tau$?**

A checked pointer checks validity in type step as rule T-CONSTC, while a tainted/unchecked pointer does not check for such during the type checking. Tainted pointers are verified through the validity check in dynamic execution as we mentioned above. **Liyi: This is saying that the validity checks and verification are the same, but the word "verification" could be misleading, so maybe change it?**

MZ: Validation instead? If for this entire section you want to emphasize what constant pointers are allowed in some specific regions, validation might be much appropriate. It might also be worth stated at the beginning that you have two kinds of validation procedures: static one for checked, and instrumented dynamic checks for unchecked region.

Liyi: I think unchecked/checked blocks and function calls below can be merged, because this section is small and might not describe the needed items.

Unchecked and Checked Blocks. During the type checking, Both $\text{checked}(\bar{x})\{e\}$ and $\text{unchecked}(\bar{x})\{e\}$ check all free variables in e are within \bar{x} ; the types for \bar{x} and the final return type τ have no checked pointers. **Liyi: why no? This should be emphasized more.** A checked or unchecked block represents the context switching from a checked to an checked region, or vice versa. We need to make sure no checked pointers are information exposed to unsafe code regions. **Liyi: example? MZ: Giving a counterexample is sometimes more convincing.**

Let Bindings and Dependent Function Pointers. Rules T-LET and T-LETINT type a let expression, which also admits type dependency. **Liyi: The type dependency means the scope of type variables. Might not need, because we have said this in previous paper.** In particular, the result of evaluating a let may have a type that refers to one of its bound variables (e.g., if the result is a checked pointer with a variable-defined bound). If so, we must substitute away this variable once it goes out of scope (T-LETINT). Note that we restrict the expression e_1 to syntactically match the structure of a Bounds expression b (see Fig. 4).

Rule T-RETINT types a ret expression when x is of type int. ret does not appear in source programs but is introduced by the semantics when evaluating a let binding (rule S-LET in Fig. 7). This rule is needed for the preservation proof. **Liyi: why?**

Rule T-FUN is the dependent function call rule. Given a function pointer type $(\text{ptr}^\xi \forall \bar{x}. \bar{\tau} \rightarrow \tau)$ from a type-check for e and the types $\bar{\tau}'$ from the argument type checks for \bar{e} , we confirm that each of $\bar{\tau}'$ is a subtype of the corresponding one in $\bar{\tau}[e'/\bar{x}]$, which replaces possible integer bound variables \bar{x} with bound expressions e' . The final result type is the defined target type τ appearing in the function pointer type also with such replacement, written as $\tau[e'/\bar{x}]$. **Liyi: should keep it uniformly with respect to the one above. It is also not so clear and too much wording here.** Consider the `safe_strcat` function in Fig. 8; its parameter type for `dst` depends on `n`. The T-FUN rule will substitute `n` with the argument at a call-site.

3.4. Type Soundness, Non-exposure, Non-crashing

MZ: Audience might be interested in how the save/load operation in tainted pointers facilitates the interaction between checked and unchecked pointer. However, I'm wondering if there's any point in the proof have something to do with this idea. If there is, it's worth talking about than listing well-formedness and consistency. Also, state the exception mechanism we are using in representing results in reduction.

In this subsection, we focus on our main meta-theoretic results about CORECHKCBOX: type soundness (progress and preservation), non-exposure, and non-crashing. These proofs have been conducted in our Coq model.

Liyi: Can we cut some? Type soundness relies on several well-formedness:

Definition 1 (Type Environment Well-formedness). A type environment Γ is well-formed if every variable mentioned as type bounds in Γ are bounded by int typed variables in Γ .

Definition 2 (Heap Well-formedness). For every m , A heap \mathcal{H} is well-formed if (i) $\mathcal{H}(m, 0)$ is undefined, and (ii) for all $n : \tau$ in the range of $\mathcal{H}(m)$, type τ contains no free variables.

Definition 3 (Stack Well-formedness). A stack snapshot φ is well-formed if for all $n : \tau$ in the range of φ , type τ contains no free variables.

We also need to introduce a notion of *consistency*, relating heap environments before and after a reduction step, and type environments, predicate sets, and stack snapshots together. **Liyi: Tell people why only checked heap/store matter?**

Definition 4 (Stack Consistency). A type environment Γ , variable predicate set Θ , and stack snapshot φ are consistent—written $\Gamma; \Theta \vdash \varphi$ —if for every variable x , $\Theta(x)$ is defined implies $\Gamma(x) = \tau$ for some τ and $\varphi(x) = n : \tau'$ for some n, τ' where $\tau' \sqsubseteq_{\Theta} \tau$.

Definition 5 (Checked Stack-Heap Consistency). A stack snapshot φ is consistent with heap \mathcal{H} —written $\mathcal{H} \vdash \varphi$ —if for every variable x , $\varphi(x) = n : \tau$ with $\text{mode}(\tau) = c$ implies $\emptyset; \mathcal{H}(c); \emptyset \vdash_c n : \tau$.

Definition 6 (Checked Heap-Heap Consistency). A heap \mathcal{H}' is consistent with \mathcal{H} —written $\mathcal{H} \triangleright \mathcal{H}'$ —if for every constant n , $\emptyset; \mathcal{H}; \emptyset \vdash_c n : \tau$ implies $\emptyset; \mathcal{H}'; \emptyset \vdash_c n : \tau$.

Progress states that a CORECHKCBOX program can always make a move:

Theorem 1 (Progress).

For any CORECHKCBOX program e , heap \mathcal{H} , stack φ , type environment Γ , and variable predicate set Θ that are all are well-formed, consistent ($\Gamma; \Theta \vdash \varphi$ and $\mathcal{H} \vdash \varphi$) and well typed ($\Gamma; \Theta \vdash_c e : \tau$ for some τ), one of the following holds:

- e is a value ($n : \tau$).
- there exists $\varphi' \mathcal{H}' r$, such that $(\varphi, \mathcal{H}, e) \rightarrow_m (\varphi', \mathcal{H}', r)$.

There are two forms of preservation regarding the checked and unchecked regions. Checked Preservation states that a reduction step preserves both the type and consistency of the program being reduced. Unchecked Preservation states that any evaluation happens at unchecked region does not affect the checked heap.

Theorem 2 (Checked Preservation). For any CORECHKCBOX program e , heap \mathcal{H} , stack φ , type environment Γ , and variable predicate set Θ that are all are well-formed, consistent ($\Gamma; \Theta \vdash \varphi$ and $\mathcal{H} \vdash \varphi$) and well typed ($\Gamma; \Theta \vdash_c e : \tau$ for some τ), if there exists φ' , \mathcal{H}' and e' , such that $(\varphi, \mathcal{H}, e) \rightarrow_c (\varphi', \mathcal{H}', e')$, then \mathcal{H}' is checked region consistent with \mathcal{H} ($\mathcal{H} \triangleright \mathcal{H}'$) and there exists Γ' and τ' that are well formed, checked region consistent ($\Gamma'; \Theta \vdash \varphi'$ and $\mathcal{H}' \vdash \varphi'$) and well typed ($\Gamma'; \Theta \vdash_c e' : \tau'$), where $\tau' \sqsubseteq_{\Theta} \tau$.

Theorem 3 (Unchecked Preservation). For any CORECHKCBOX program e , heap \mathcal{H} , stack φ , type environment Γ , and variable predicate set Θ that are all are well-formed and well typed ($\Gamma; \Theta \vdash_c e : \tau$ for some τ), if there exists φ' , \mathcal{H}' and e' , such that $(\varphi, \mathcal{H}, e) \rightarrow_u (\varphi', \mathcal{H}', e')$, then $\mathcal{H}'(c) = \mathcal{H}(c)$.

Liyi: maybe some intuition why this is important.

Using the above theorem, we first show the non-exposure theorem, where code in unchecked region cannot observe a valid checked pointer address.

Theorem 4 (Non-Exposure). For any CORECHKCBOX program e , heap \mathcal{H} , stack φ , type environment Γ , and variable predicate set Θ that are all are well-formed and well typed ($\Gamma; \Theta \vdash_c e : \tau$ for some τ), if there exists φ' , \mathcal{H}' and e' , such that $(\varphi, \mathcal{H}, e) \rightarrow_u (\varphi', \mathcal{H}', e')$ and $e = E[\alpha(x)]$ and $\text{mode}(E) = u$, where $\alpha(x)$ is some expression (not checked nor unchecked) containing variable x ; thus, it is not a checked pointer.

We now state our main result, *non-crashing*, which suggests that a well-typed program can never be *stuck* (expression e is a non-value that cannot take a step³).

Theorem 5 (Non-Crashing). For any CORECHKCBOX program e , heap \mathcal{H} , stack φ , type environment Γ , and variable predicate set Θ that are well-formed and consistent ($\Gamma; \Theta \vdash \varphi$ and $\mathcal{H} \vdash \varphi$), if e is well-typed ($\varphi; \Theta \vdash_c e : \tau$ for some τ) and there exists φ_i , \mathcal{H}_i , e_i , and m_i for $i \in [1, k]$, such that $(\varphi, \mathcal{H}, e) \rightarrow_{m_1} (\varphi_1, \mathcal{H}_1, e_1) \rightarrow_{m_2} \dots \rightarrow_{m_k} (\varphi_k, \mathcal{H}_k, r)$, then r can never be *stuck*.

MZ: It's in the footnote. But, we should explain the definition of "stuck" in the text. It's not crystal clear.

3. Note that bounds and null are *not* stuck expressions—they represent a program terminated by a failed run-time check. A program that tries to access $\mathcal{H}n$ but \mathcal{H} is undefined at n will be stuck, and violates spatial safety.

	c	t	u
	CBOX / CORE	CBOX / CORE	CBOX / CORE
c	$*x / *(c, x)$	$\text{sand_get}(x) / *(u, x)$	\times
u	\times	$*x / *(u, x)$	$*x / *(u, x)$

Figure 12: Compiled Targets for Dereference

4. Compilation

The main subtlety of compiling Checked C to Clang/L-LVM is to capture the annotations on pointer literals that track array bound information, which is used in premises of rules like S-DEFARRAY and S-ASSIGNARR to prevent spatial safety violations. The Checked C compiler [16] inserted additional pointer checks for verifying pointers are not null and the bounds are within their limits. **Le: what does it mean by bounds within limits?** The latter is done by introducing additional shadow variables for storing (NT-)array pointer bound information.

In CHECKEDCBOX, context and pointer modes determine the particular heap/function store that a pointer points to, i.e., c pointers point to checked regions, while t and u pointers point to unchecked regions. **Le: context mode doesn't determine where pointer points to but check if it is valid to point to that region?** Unchecked regions are associated with a sandbox mechanism that permits exception handling of potential memory failures. In the compiled LLVM code, pointer access operations have different syntaxes when the modes are different. **Le: should we explain why we use different syntax instead of compiling the corresponding heap to llvm** Figure 12 lists the different compiled syntaxes of a dereference operation ($*x$) for the compiler implementation (CBOX, stands for CHECKEDCBOX) and formalism (CORE, stands for CORECHKCBOX). The columns represent different pointer modes and the rows represent context modes. For example, when we have a t-mode pointer in a c-mode region, we compile a dereference operation to the sandbox pointer access function ($\text{sand_get}(x)$) accessing the data in the CHECKEDCBOX implementation. In CORECHKCBOX, we create a new dereference data-structure on top of the existing $*x$ operation (in LLVM): $*(m, x)$. If the mode is c, it accesses the checked heap/function store; otherwise, it accesses the unchecked one.

This section shows how CORECHKCBOX deals with pointer modes, mode switching and function pointer compilations, with no loss of expressiveness as the Checked C contains the erase of annotations in [16] and Appendix G. For the compiler formalism, we present a compilation algorithm that converts from CORECHKCBOX to COREC, an untyped language without metadata annotations, which represents an intermediate layer we build on LLVM for simplifying compilation. In COREC, the syntax for dereference, assignment, malloc, function calls are: $*(m, e)$, $*(m, e) = e$, $\text{malloc}(m, \omega)$, and $(m, e)(\bar{e})$. **Le: strlen?** The algorithm sheds light on how compilation can be implemented in the real Checked C compiler, while eschewing many vital details (COREC has many differences with LLVM IR).

Compilation is defined by extending CORECHKCBOX's typing judgment as follows:


$$\Gamma; \Theta; \rho \vdash_m e \gg \dot{e} : \tau$$

There is now a COREC output \dot{e} and an input ρ , which maps each (NT-)array pointer variable to its mode and each variable p to a pair of *shadow variables* that keep p 's up-to-date upper and lower bounds. These may differ from the bounds in p 's type due to bounds widening.⁴ **Le: These two sentences are confusing.**

We formalize rules for this judgment in PLT Redex [9], following and extending our Coq development for CORECHKCBOX. To give confidence that compilation is correct, we use Redex's property-based random testing support to show that compiled-to \dot{e} simulates e , for all e .

4.1. Approach

Due to space constraints, we explain the rules for compilation by examples, using a C-like syntax; the complete rules are given in Appendix G. Each rule performs up to three tasks: (a) conversion of e to A-normal form; (b) insertion of dynamic checks and bound widening expressions; and (c) generate right pointer accessing expressions based on modes. A-normal form conversion is straightforward: compound expressions are managed by storing results of subexpressions into temporary variables, as in the following example.

```
let y=(x+1)+(6+1);
      
      let a=x+1;
      let b=6+1;
      let y=a+b
```


This simplifies the management of effects from subexpressions. The next two steps of compilation are more interesting. We state them based on different CORECHKCBOX operations.

Pointer Accesses and Modes. In every declaration **Le: (or the beginning of a function body) why?** of a pointer, if the pointer is an (NT-)array, we first allocate two *shadow variables* to track the lower and upper bounds which are potentially changed for pointer arithmetic and NT-array bound widening. Each c-mode NT-array pointer variable is associated with its type information in a store. Additionally, we place bounds and null-pointer checks, such as the line 6 and 7 in Figure 13. In addition, in the formalism, before every use of a tainted pointer (Figure 13 line 9 and 10), there is an inserted verification step similar to Figure 11, which checks if a pointer is well defined in the heap (**not_null**) and the spatial safety. Predicate **not_null** checks that every element in the pointer's range (**p_lo** and **p_hi**) is well defined in the heap. The modes in compiled deference ($*(\text{mode}(p) \wedge m, p)$) and assignment ($*(\text{mode}(q) \wedge m, q)=1$) operations are computed based on the meet operation (\wedge) of the pointer mode (e.g. **mode(p)**) and the current context mode (**m**).

4. Since lower bounds are never widened, the lower-bound shadow variable is unnecessary; we include it for uniformity.

```
1 int deref_array(n : int,
2   p : ptrc [(0,n) int]nt,
3   q : ptrt [(0,n) int]nt) {
4   /* ρ(p) = p_lo, p_hi, p_m */
5   /* ρ(q) = q_lo, q_hi, q_m */
6   * p;
7   * q = 1;
8 }
9 ...
10 /* p0 : ptrc [(0,5) int]nt */
11 /* q0 : ptrt [(0,5) int]nt */
12 deref_array(5, p0, q0);

```



```
1 deref_array(int n, int* p, int * q) {
2   //m is the current context mode
3   let p_lo = 0; let p_hi = n;
4   let q_lo = 0; let q_hi = n;
5   /* runtime checks */
6   assert(p_lo ≤ 0 && 0 ≤ p_hi);
7   assert(p != 0);
8   *(mode(p) ∧ m, p);
9   verify(q, not_null(m, q_lo, q_hi)
10    && q_lo ≤ 0 && 0 ≤ q_hi);
11   *(mode(q) ∧ m, q)=1;
12 }
13 ...
14 deref_array(5, p0, q0);

```

Figure 13: Compilation Example for Dependent Functions

Checked and Unchecked Blocks. In the CHECKED-CBOX implementation, unchecked and checked blocks are compiled as context switching functions provided by the sandbox mechanism. $\text{unchecked}(\bar{x})\{e\}$ is compiled to $\text{sandbox_call}(\bar{x}, e)$, where we call the sandbox to execute expression e with the arguments \bar{x} . $\text{checked}(\bar{x})\{e\}$ is compiled to $\text{callback}(\bar{x}, e)$, where we perform a callback to a checked block code e inside a sandbox. In CHECKEDCBOX, we adopt an aggressive execution scheme that directly learns pointer addresses from compiled assembly to make the callback happen. In the formalism, we rely on the type system to guarantee the context switching without creating the extra function calls for simplicity.

Function Pointers and Calls. Function pointers are managed similarly to normal pointers, but we insert checks to check if the pointer address is not null in the function store instead of heap, and whether or not the type is correctly represented, for both c and t mode pointers⁵. For example, in compiling the **stringsort** function in Figure 1, we place a check **verify_fun(cmp, not_null(c, p_lo, p_hi) && type_match)**, and we place a similar check before Figure 3 line 7 to check the tainted **cmp** when it is used. The compilation of function calls (compiling to $(m, e)(\bar{e})$) is similar to the manipulation of pointer access operations in Figure 12.

5. c-mode pointers are checked once in the beginning and t-mode pointers are checked every time when use

For compiling dependent function calls, Figure 13 provides a hint. Notice that the bounds for the array pointer `p` are not passed as arguments. Instead, they are initialized according to `p`'s type—see line 4 of the original CORECHKCBOX program at the top of the figure. Line 3 of the generated code sets the lower bound to 0 and the upper bound to `n`.

4.2. Metatheory

We formalize both the compilation procedure and the simulation theorem in the PLT Redex model we developed for CORECHKCBOX (see Sec. 3.1), and then attempt to falsify it via Redex's support for random testing. Redex allows us to specify compilation as logical rules (an extension of typing), but then execute it algorithmically to automatically test whether simulation holds. This process revealed several bugs in compilation and the theorem statement. We ultimately plan to prove simulation in the Coq model.

We use the notation \gg to indicate the *erasure* of stack and heap—the rhs is the same as the lhs but with type annotations removed:

$$\begin{aligned} \mathcal{H} &\gg \dot{\mathcal{H}} \\ \varphi &\gg \dot{\varphi} \end{aligned}$$

In addition, when $\Gamma; \emptyset \vdash \varphi$ and φ is well-formed, we write $(\varphi, \mathcal{H}, e) \gg_m (\dot{\varphi}, \dot{\mathcal{H}}, \dot{e})$ to denote $\varphi \gg \dot{\varphi}$, $\mathcal{H} \gg \dot{\mathcal{H}}$ and $\Gamma; \emptyset; \emptyset \vdash_m e \gg \dot{e} : \tau$ for some τ respectively. Γ is omitted from the notation since the well-formedness of φ and its consistency with respect to Γ imply that e must be closed under φ , allowing us to recover Γ from φ . Finally, we use $\dot{\rightarrow}^*$ to denote the transitive closure of the reduction relation of COREC. Unlike the CORECHKCBOX, the semantics of COREC does not distinguish checked and unchecked regions.

Fig. 14 gives an overview of the simulation theorem.⁶ The simulation theorem is specified in a way that is similar to the one by Merigoux et al. [22].

An ordinary simulation property would replace the middle and bottom parts of the figure with the following:

$$(\dot{\varphi}_0, \dot{\mathcal{H}}_0, \dot{e}_0) \dot{\rightarrow}^* (\dot{\varphi}_1, \dot{\mathcal{H}}_1, \dot{e}_1)$$

Instead, we relate two erased configurations using the relation \sim , which only requires that the two configurations will eventually reduce to the same state.

Theorem 6 (Simulation (\sim)). For CORECHKCBOX expressions e_0 , stacks φ_0, φ_1 , and heap snapshots $\mathcal{H}_0, \mathcal{H}_1$, if $\mathcal{H}_0 \vdash \varphi_0$, $(\varphi_0, \mathcal{H}_0, e_0) \gg_c (\dot{\varphi}_0, \dot{\mathcal{H}}_0, \dot{e}_0)$, and if there exists some r_1 such that $(\varphi_0, \mathcal{H}_0, e_0) \rightarrow_c (\varphi_1, \mathcal{H}_1, r_1)$, then the following facts hold:

- if there exists e_1 such that $r = e_1$ and $(\varphi_1, \mathcal{H}_1, e_1) \gg (\dot{\varphi}_1, \dot{\mathcal{H}}_1, \dot{e}_1)$, then there exists some $\dot{\varphi}, \dot{\mathcal{H}}, \dot{e}$, such that $(\dot{\varphi}_0, \dot{\mathcal{H}}_0, \dot{e}_0) \dot{\rightarrow}^* (\dot{\varphi}, \dot{\mathcal{H}}, \dot{e})$ and $(\dot{\varphi}_1, \dot{\mathcal{H}}_1, \dot{e}_1) \dot{\rightarrow}^* (\dot{\varphi}, \dot{\mathcal{H}}, \dot{e})$.

6. We ellide the possibility of \dot{e}_1 evaluating to bounds or null in the diagram for readability.

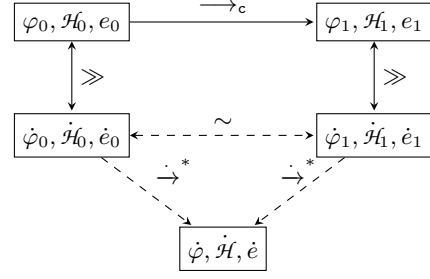


Figure 14: Simulation between CORECHKCBOX and COREC

- if $r_1 = \text{bounds or null}$, then we have $(\dot{\varphi}_0, \dot{\mathcal{H}}_0, \dot{e}_0) \dot{\rightarrow}^* (\dot{\varphi}_1, \dot{\mathcal{H}}_1, r_1)$ where $\varphi_1 \gg \dot{\varphi}_1$, $\mathcal{H}_1 \gg \dot{\mathcal{H}}_1$.

As our random generator never generates unchecked expressions (whose behavior could be undefined), we can only test the simulation theorem as it relates to checked code. This limitation makes it unnecessary to state the other direction of the simulation theorem where e_0 is stuck, because Theorem 1 guarantees that e_0 will never enter a stuck state if it is well-typed in checked mode.

The current version of the Redex model has been tested against 23000 expressions with depth less than 11. Each expression can reduce multiple steps, and we test simulation between every two adjacent steps to cover a wider range of programs, particularly the ones that have a non-empty heap.

5. Evaluation

- provide evidence that the CHECKEDCBOX compiler is efficient. Compare the compiler with respect to other work, like RLBox, also the previous Checked C compiler.
- provide user experience of CHECKEDCBOX. We restrict the use of checked pointers compared to previous checked-c compiler. Is the restriction arragable. We can say that the tainted shells are auto-matically generated, so we have a mechanism for auto-generating tainted pointers if necessary.
- if we have space, we can re-introduce random testing a little, saying that how it helps us to develop the compiler.
- we can then talk about the possible bugs we find in the Checked C compiler for function pointer or the RLBox bugs.

6. Related Work

Our work is most closely related to prior formalizations of C(-like) languages and program partitioning mechanism that aim at enforcing memory safety, but it also touches on C-language formalization in general.

Formalizing C and Low-level code. A number of prior works have looked at formalizing the semantics of C, including CompCert [2, 15], Ellison and Rosu [8], Kang et al.

[13], and Memarian et al. [20, 21]. These works also model pointers as logically coupled with either the bounds of the blocks they point to, or provenance information from which bounds can be derived. None of these is directly concerned with enforcing spatial safety, and that is reflected in the design. For example, memory itself is not represented as a flat address space, as in our model or real machines, so memory corruption due to spatial safety violations, which Checked C’s type system aims to prevent, may not be expressible. That said, these formalizations consider much more of the C language than does CORECHKCBOX, since they are interested in the entire language’s behavior.

Spatially Safe C Formalizations. Several prior works formalize C-language transformations or C-language dialects aiming to ensure spatial safety. Hathhorn et al. [11] extended the formalization of Ellison and Rosu [8] to produce a semantics that detects violations of spatial safety (and other forms of undefinedness). It uses a CompCert-style memory model, but “fattens” logical pointer representations to facilitate adding side conditions similar to CORECHKCBOX’s. Its concern is bug finding, not compiling programs to use this semantics.

CCured [25] and Softbound [23] implement spatially safe semantics for normal C via program transformation. Like CORECHKCBOX, both systems’ operational semantics annotate pointers with their bounds. CCured’s equivalent of array pointers are compiled to be “fat,” while SoftBound compiles bounds metadata to a separate hashtable, thus retaining binary compatibility at higher checking cost. Checked C uses static type information to enable bounds checks without need of pointer-attached metadata, as we show in Section 4. Neither CCured nor Softbound models null-terminated array pointers, whereas our semantics ensures that such pointers respect the zero-termination invariant, leveraging bounds widening to enhance expressiveness.

Cyclone [10, 12] is a C dialect that aims to ensure memory safety; its pointer types are similar to CCured. Cyclone’s formalization [10] focuses on the use of *regions* to ensure temporal safety; it does not formalize arrays or threats to spatial safety. Deputy [5, 35] is another safe-C dialect that aims to avoid fat pointers. It was an initial inspiration for Checked C’s design [7], though it provides no specific modeling for null-terminated array pointers. Deputy’s formalization [5] defines its semantics directly in terms of compilation, similar in style to what we present in Section 4. Doing so tightly couples typing, compilation, and semantics, which are treated independently in CORECHKCBOX. Separating semantics from compilation isolates meaning from mechanism, easing understandability. Indeed, it was this separation that led us to notice the limitation with Checked C’s handling of bounds widening.

The most closely related work is the formalization of Checked C done by Ruef et al. [26] and Li et al. [16]. They presented the type system and semantics of a core model of Checked C, mechanized in Coq, and prove a blame theorem. The difference between these works and CHECKEDCBOX is listed in Section 1 and Section 2.

Program Partitioning Mechanism. The unchecked and

checked code region separation in CHECKEDCBOX represents an isolation mechanism to ensure that code executed as part of unchecked regions does not violate the safety guarantees in checked regions, which is typically called program partitioning [27], and there has been considerable work [1, 4, 17, 18, 31] in the area. Most of these techniques are *data-centric* [17, 18], wherein program data drives the partitioning. E.g., Given sensitive data in a program, the goal is to partition functions into two parts or partitions based on whether a function can access the sensitive data. The performance overhead of these approaches is dominated by marshaling costs and depends on the usage of sensitive data. The overhead of state-of-the-art approaches [17, 18] is prohibitive and varies from 37%-163%.

Narayan et al. [24], a.k.a RLBox, merged a type checker with a sandbox mechanism to better achieve the programming partitioning mechanism. They allow tainted pointers to be shared among different code regions. There are several differences between their work with CHECKEDCBOX.

- There is no RLBox formalism.
- RLBox implements program partition without spatial safety guarantee, while CHECKEDCBOX provides the non-crashing guarantee.
- RLBox’s type system is based on C++ templates, which might contain potential unknown faulty.
- RLBox only provides program partition for third party library functions, while CHECKEDCBOX can context-switch between checked and unchecked code regions for arbitrary functions.
- RLBox has no non-exposure guarantee, a checked pointer might leak its pointer address in a untrusted third party library function, while CHECKEDCBOX ensures that checked pointers cannot be leaked to unchecked code regions.

7. Conclusion and Future Work

This paper presented CORECHKCBOX, a formalization of an extended core of the Checked C language which aims to provide spatial memory safety. CORECHKCBOX models dynamically sized and null-terminated arrays with dependently typed bounds that can additionally be widened at runtime. We prove, in Coq, the key safety property of Checked C for our formalization, *blame*: if a mix of checked and unchecked code gives rise to a spatial memory safety violation, then this violation originated in an unchecked part of the code. We also show how programs written in CORECHKCBOX (whose semantics leverage fat pointers) can be compiled to COREC (which does not) while preserving their behavior. We developed a version of CORECHKCBOX written in PLT Redex, and used a custom term generator in conjunction with Redex’s randomized testing framework to give confidence that compilation is correct. We also used this framework to cross-check CORECHKCBOX against the Checked C compiler, finding multiple inconsistencies in the process.

As future work, we wish to extend CORECHKCBOX to model more of Checked C, with our Redex-based test-

ing framework guiding the process. The most interesting Checked C feature not yet modeled is *interop types* (itypes), which are used to simplify interactions with unchecked code via function calls. A function whose parameters are itypes can be passed checked or unchecked pointers depending on whether the caller is in a checked region. This feature allows for a more modular C-to-Checked C porting process, but complicates reasoning about blame. A more ambitious next step would be to extend an existing formally verified framework for C, such as CompCert [14] or VeLLVM [34], with Checked C features, towards producing a verified-correct Checked C compiler. We believe that CORECHKCBOX’s Coq and Redex models lay the foundation for such a step, but substantial engineering work remains.

References

- [1] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. *USENIX Association*, 2008.
- [2] Sandrine Blazy and Xavier Leroy. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. ISSN 1573-0670. doi: 10.1007/s10817-009-9148-3. URL <http://dx.doi.org/10.1007/s10817-009-9148-3>.
- [3] BlueHat. Memory corruption is still the most prevalent security vulnerability. <https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/>, 2019. Accessed: 2020-02-11.
- [4] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, volume 57, 2004.
- [5] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent Types for Low-Level Programming. In *Proceedings of European Symposium on Programming (ESOP ’07)*, 2007.
- [6] Junhan Duan, Yudi Yang, Jie Zhou, and John Criswell. Refactoring the FreeBSD kernel with Checked C. In *Proceedings of the 2020 IEEE Cybersecurity Development Conference (SecDev)*, 2020.
- [7] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 53–60, 2018. doi: 10.1109/SecDev.2018.00015.
- [8] Chucky Ellison and Grigore Rosu. An Executable Formal Semantics of C with Applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’12*, pages 533–544, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103719. URL <http://doi.acm.org/10.1145/2103656.2103719>.
- [9] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009. ISBN 0262062755.
- [10] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based Memory Management in Cyclone. In *PLDI*, 2002.
- [11] Chris Hathhorn, Chucky Ellison, and Grigore Rosu. Defining the Undefinedness of C. *SIGPLAN Not.*, 50(6):336–345, June 2015. ISSN 0362-1340. doi: 10.1145/2813885.2737979. URL <https://doi.org/10.1145/2813885.2737979>.
- [12] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, , and Yanling Wang. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, 2002. USENIX.
- [13] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A Formal C Memory Model Supporting Integer-pointer Casts. *SIGPLAN Not.*, 50(6):326–335, June 2015. ISSN 0362-1340. doi: 10.1145/2813885.2738005. URL <http://doi.acm.org/10.1145/2813885.2738005>.
- [14] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009. ISSN 0001-0782. doi: 10/c9sb7q. URL <http://doi.acm.org/10.1145/1538788.1538814>.
- [15] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA, June 2012. URL <https://hal.inria.fr/hal-00703441>.
- [16] Liyi Li, Yiyun Liu, Deena L. Postol, Leonidas Lampropoulos, David Van Horn, and Michael Hicks. A formal model of Checked C. In *Proceedings of the Computer Security Foundations Symposium (CSF)*, August 2022.
- [17] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O’Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eysers, Rüdiger Kapitza, et al. Glamdring: Automatic application partitioning for intel {SGX}. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298, 2017.
- [18] Shen Liu, Gang Tan, and Trent Jaeger. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2359–2371, 2017.
- [19] Aravind Machiry, John Kastner, Matt McCutchen, Aaron Eline, Kyle Headley, and Michael Hicks. C to checked c by 3c. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–29, 2022.
- [20] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyn-dylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the Depths of C: Elaborating the de Facto Standards. *SIGPLAN Not.*, 51(6):1–15, June 2016. ISSN 0362-1340. doi: 10.1145/2980983.2908081. URL <https://doi.org/10.1145/2980983.2908081>.
- [21] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C Semantics and Pointer Provenance. *Proc. ACM Program. Lang.*, 3(POPL):67:1–67:32, January 2019. ISSN 2475-1421. doi: 10.1145/3290380. URL <http://doi.acm.org/10.1145/3290380>.
- [22] Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. Catala: A Programming Language for the Law. *arXiv preprint arXiv:2103.03198*, 2021.
- [23] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’09*, page 245–258, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. doi: 10.1145/1542476.1542504. URL <https://doi.org/10.1145/1542476.1542504>.
- [24] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 699–716. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL <https://www.usenix.org/conference/usenixsecurity20/presentation/narayan>.

- [25] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3), 2005.
- [26] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. Achieving Safety Incrementally with Checked C. In Flemming Nielson and David Sands, editors, *Principles of Security and Trust*, pages 76–98, Cham, 2019. Springer International Publishing. ISBN 978-3-030-17138-4.
- [27] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. Towards automatic program partitioning. In *Proceedings of the 6th ACM conference on Computing frontiers*, pages 89–98, 2009.
- [28] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, 2012.
- [29] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS. ACM*, 2004.
- [30] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for security. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (S&P)*, 2019.
- [31] Gang Tan et al. Principles and implementation techniques of software-based fault isolation. *Foundations and Trends® in Privacy and Security*, 1(3):137–198, 2017.
- [32] CVE Trends. Cve trends. <https://www.cvedetails.com/vulnerabilities-by-types.php>, 2021. Accessed: 2020-10-11.
- [33] Bin Zeng, Gang Tan, and Úlfar Erlingsson. Strato: A Retargetable Framework for Low-level Inlined-reference Monitors. In *Proceedings of the 22Nd USENIX Conference on Security*, 2013.
- [34] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. *SIGPLAN Not.*, 47(1):427–440, January 2012. ISSN 0362-1340. doi: 10.1145/2103621.2103709. URL <http://doi.acm.org/10.1145/2103621.2103709>.
- [35] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *7th Symposium on Operating System Design and Implementation (OSDI'06)*, Seattle, Washington, 2006. USENIX Association.

Appendix

1. Differences with the Coq and Redex Models

The Coq and Redex models of CORECHKCBOX may be found at <https://github.com/plum-umd/checkedc>. The Coq model’s syntax is slightly different from that in Fig. 4. In particular, the arguments in a function call are restricted to variables and constants, according to a separate well-formedness condition. A function call $f(e)$ can always be written in `let x = e in f(x)` to cope. In addition, conditionals have two syntactic forms: `EIf` is a normal conditional, and `ElfDef` is one whose boolean guard is of the form $*x$. By syntactically distinguishing these two cases, the Coq model does not need the *[prefer]* rule for `if (*x)...` forms as in Fig. 6. The Redex model *does* prioritize such forms but not the same way as in the figure. It uses a variation of the S-VAR rule: The modified rule is equipped with a precondition that is false whenever S-IFNTT is applicable.

The Coq model uses a runtime stack φ as described at the start of Sec. 3.2. The Redex model introduces let bindings during evaluation to simulate a runtime stack. For example, consider the expression $e \equiv \text{let } x = (5 : \text{int}) \text{ in } x + x$. Expression e first steps to `let x = (5 : int) in (5:int) + x`, which in turns steps to `let x = (5 : int) in (5:int) + (5:int)`. Since the rhs of x is a value, the let binding in e effectively functions as a stack that maps from x to $5 : \text{int}$. The let form remains in the expression and lazily replaces the variables in its body. The let form can be removed from the expression only if its body is evaluated to a value, e.g., `let x = (5 : int) in (10:int)` steps to `10:int`. The rule for popping let bindings in this manner corresponds to the S-RET rule in Fig. 7. Leveraging let bindings adds complexity to the semantics but simplifies typing/consistency and term generation during randomized testing.

2. Typing Rules for Literal Pointers

The typing of integer literals, which can also be pointers to the heap, was presented in Sec. 3.4 in Fig. 11. Here we describe these rules further.

The variable type rule (T-VAR) simply checks if a given variable has the defined type in Γ ; the constant rule (T-CONST) is slightly more involved. First, it ensures that the type annotation τ does not contain any free variables. More importantly, it ensures that the literal itself is well typed using an auxilliary typing relation $\mathcal{H}; \sigma \vdash n : \tau$.

If the literal’s type is an integer, an unchecked pointer, or a null pointer, it is well typed, as shown by the top three rules in Fig. 11. However, if it is a checked pointer $\text{ptr}^c \omega$, we need to ensure that what it points to in the heap is of the appropriate pointed-to type (ω), and also recursively ensure that any literal pointers reachable this way are also well-typed. This is captured by the bottom rule in the figure, which states that for every location $n + i$ in the pointers’ range $[n, n + \text{size}(\omega))$, where size yields the size of its

argument, then the value at the location $\mathcal{H}(n + i)$ is also well-typed. However, as heap snapshots can contain cyclic structures (which would lead to infinite typing derivations), we use a scope σ to assume that the original pointer is well-typed when checking the types of what it points to. The middle rule then accesses the scope to tie the knot and keep the derivation finite, just like in Ruef et al. [26].

3. Other Semantic Rules

Fig. 15 shows the remaining semantic rules for CORECHKBOX. We explain a selected few rules in this subsection.

Rule S-VAR loads the value for x in stack φ . Rule S-DEFARRAY dereferences an array pointer, which is similar to the Rule S-DEFNTARRAY in Fig. 7 (dealing with null-terminated array pointers). The only difference is that the range of 0 is at $[n_l, n_h)$ not $[n_l, n_h]$, meaning that one cannot dereference the upper-bound position in an array. Rules DEFARRAYBOUND and DEFNTARRAYBOUND describe an error case for a dereference operation. If we are dereferencing an array/NT-array pointer and the mode is c, 0 must be in the range from n_l to n_h (meaning that the dereference is in-bound); if not, the system results in a bounds error. Obviously, the dereference of an array/NT-array pointer also experiences a null state transition if $n \leq 0$.

Rules S-MALLOC and S-MALLOCBOUND describe the malloc semantics. Given a valid type ω_a that contains no free variables, alloc function returns an address pointing at the first position of an allocated space whose size is equal to the size of ω_a , and a new heap snapshot \mathcal{H}' that marks the allocated space for the new allocation. The malloc is transitioned to the address n with the type $\text{ptr}^c \omega_a$ and new updated heap. It is possible for malloc to transition to a bounds error if the ω_a is an array/NT-array type $[(n_l, n_h) \tau]_\kappa$, and either $n_l \neq 0$ or $n_h \leq 0$. This can happen when the bound variable is evaluated to a bound constant that is not desired.

4. Subtyping for dependent types

The subtyping relation given in Fig. 10 involves dependent bounds, i.e., bounds that may refer to variables. To decide premises $b \leq b'$, we need a decision procedure that accounts for the possible values of these variables. This process considers Θ , tracked by the typing judgment, and φ , the current stack snapshot (when performing subtyping as part of the type preservation proof).

Definition 7 (Inequality).

- $n \leq m$ if n is less than or equal to m .
- $x + n \leq x + m$ if n is less than or equal to m .
- All other cases result in false.

To capture bound variables in dependent types, the Checked C subtyping relation (\sqsubseteq) is parameterized by a restricted stack snapshot $\varphi|_\rho$ and the predicate map Θ , where φ is a stack and ρ is a set of variables. $\varphi|_\rho$ means to restrict the domain of φ to the variable set ρ . Clearly, we have the

relation: $\varphi|_\rho \sqsubseteq \varphi$. \sqsubseteq being parameterized by $\varphi|_\rho$ refers to that when we compare two bounds $b \leq b'$, we actually do $\varphi|_\rho(b) \leq \varphi|_\rho(b')$ by interpreting the variables in b and b' with possible values in $\varphi|_\rho$. Let's define a subset relation \preceq for two restricted stack snapshot $\varphi|_\rho$ and $\varphi'|_\rho$:

Definition 8 (Subset of Stack Snapshots). Given two $\varphi|_\rho$ and $\varphi'|_\rho$, $\varphi|_\rho \preceq \varphi'|_\rho$, iff for $x \in \rho$ and y , $(x, y) \in \varphi|_\rho \Rightarrow (x, y) \in \varphi'|_\rho$.

For every two restricted stack snapshots $\varphi|_\rho$ and $\varphi'|_\rho$, such that $\varphi|_\rho \preceq \varphi'|_\rho$, we have the following theorem in Checked C (proved in Coq):

Theorem 7 (Stack Snapshot Theorem). Given two types τ and τ' , two restricted stack snapshots $\varphi|_\rho$ and $\varphi'|_\rho$, if $\varphi|_\rho \preceq \varphi'|_\rho$, and $\tau \sqsubseteq \tau'$ under the parameterization of $\varphi|_\rho$, then $\tau \sqsubseteq \tau'$ under the parameterization of $\varphi'|_\rho$.

Clearly, for every $\varphi|_\rho$, we have $\emptyset \preceq \varphi|_\rho$. The type checking stage is a compile-time process, so $\varphi|_\rho$ is \emptyset at the type checking stage. Stack snapshots are needed for proving type preserving, as variables in bounds expressions are evaluated away.

As mentioned in the main text, \sqsubseteq is also parameterized by Θ , which provides the range of allowed values for a bound variable; thus, more \sqsubseteq relation is provable. For example, in Fig. 8, the `strlen` operation in line 4 turns the type of `dst` to be $\text{ptr}^c [(0, x) \text{int}]_{nt}$ and extends the upper bound to `x`. In the `strlen` type rule, it also inserts a predicate `x ≥ 0` in Θ ; thus, the cast operation in line 16 is valid because $\text{ptr}^c [(0, x) \text{int}]_{nt} \sqsubseteq \text{ptr}^c [(0, 0) \text{int}]_{nt}$ is provable when we know `x ≥ 0`.

Note that if φ and Θ are \emptyset , we do only the syntactic \leq comparison; otherwise, we apply φ to both sides of \sqsubseteq , and then determine the \leq comparison based on a Boolean predicate decision procedure on top of Θ . This process allows us to type check both an input expression and the intermediate expression after evaluating an expression.

5. Other Type Rules

Here we show the type rules for other Checked C operations in Fig. 16. Rule T-DEF is for dereferencing a non-array pointer. The statement $m \leq m'$ ensures that no unchecked pointers are used in checked regions. Rule T-MAC deals with malloc operations. There is a well-formedness check to require that the possible bound variables in ω must be in the domain of Γ (see Fig. 18). This is similar to the well-formedness assumption of the type environment (Definition 1) Rule T-ADD deals with binary operations whose subterms are integer expressions, while rule T-IND serves the case for pointer arithmetic. For simplicity, in the Checked C formalization, we do not allow arbitrary pointer arithmetic. The only pointer arithmetic operations allowed are the forms shown in rules T-IND and T-INDASSIGN in Fig. 16. Rule T-ASSIGN assigns a value to a non-array pointer location. The predicate $\tau' \sqsubseteq \tau$ requires that the value being assigned is a subtype of the pointer type. The T-INDASSIGN rule is an extended assignment operation for handling assignments

$$\begin{array}{c}
\text{S-VAR} \\
(\varphi, \mathcal{H}, x) \longrightarrow (\varphi, \mathcal{H}, \varphi(x)) \\
\\
\text{S-DEFARRAY} \\
\frac{\mathcal{H}(n) = n_a : \tau_a \quad 0 \in [n_l, n_h]}{(\varphi, \mathcal{H}, * n : \text{ptr}^c [(n_l, n_h) \tau]_{nt}) \longrightarrow (\varphi, \mathcal{H}, n_a : \tau)} \\
\\
\text{S-DEFARRAYBOUND} \\
\frac{0 \notin [n_l, n_h]}{(\varphi, \mathcal{H}, * n : \text{ptr}^c [(n_l, n_h) \tau]_{\kappa}) \longrightarrow (\varphi, \mathcal{H}, \text{bounds})} \\
\\
\text{S-DEFNTARRAYBOUND} \\
\frac{0 \notin [n_l, n_h]}{(\varphi, \mathcal{H}, * n : \text{ptr}^c [(n_l, n_h) \tau]_{nt}) \longrightarrow (\varphi, \mathcal{H}, \text{bounds})} \\
\\
\text{S-ASSIGN} \\
\frac{\mathcal{H}(n) = n_a : \tau_a}{(\varphi, \mathcal{H}, * n : \text{ptr}^c \tau = n_1 : \tau_1) \longrightarrow (\varphi, \mathcal{H}[n \mapsto n_1 : \tau], n_1 : \tau)} \\
\\
\text{S-ASSIGNNULL} \\
(\varphi, \mathcal{H}, * 0 : \text{ptr}^c \omega = n_1 : \tau_1) \longrightarrow (\varphi, \mathcal{H}, \text{null}) \\
\\
\text{S-ASSIGNARRBOUND} \\
\frac{0 \notin [n_l, n_h]}{(\varphi, \mathcal{H}, * n : \text{ptr}^c [(n_l, n_h) \tau]_{\kappa} = n_1 : \tau_1) \longrightarrow (\varphi, \mathcal{H}, \text{bounds})} \\
\\
\text{S-MALLOC} \\
\frac{\varphi(\omega) = \omega_a \quad \text{alloc}(\mathcal{H}, \omega_a) = (n, \mathcal{H}')}{(\varphi, \mathcal{H}, \text{malloc}(\omega,)) \longrightarrow (\varphi, \mathcal{H}', n : \text{ptr}^c \omega_a)} \\
\\
\text{S-MALLOCBOUND} \\
\frac{\varphi(\omega) = [(n_l, n_h) \tau]_{\kappa} \quad (n_l \neq 0 \vee n_h \leq 0)}{(\varphi, \mathcal{H}, \text{malloc}(\omega,)) \longrightarrow (\varphi, \mathcal{H}', \text{bounds})} \\
\\
\text{S-IFT} \\
\frac{n \neq 0}{(\varphi, \mathcal{H}, \text{if } (n : \tau) e_1 \text{ else } e_2) \longrightarrow (\varphi, \mathcal{H}, e_1)} \\
\\
\text{S-IFFF} \\
(\varphi, \mathcal{H}, \text{if } (0 : \tau) e_1 \text{ else } e_2) \longrightarrow (\varphi, \mathcal{H}, e_2) \\
\\
\text{S-UNCHECKED} \\
(\varphi, \mathcal{H}, \text{unchecked}(n : \tau) \{ \longrightarrow \}) \longrightarrow (\varphi, \mathcal{H}, n : \tau) \\
\\
\text{S-STR} \\
\frac{0 \in [n_l, n_h] \quad n_a \leq n_h \quad \mathcal{H}(n + n_a) = 0 \quad (\forall i. n \leq i < n + n_a \Rightarrow (\exists n_i t_i. \mathcal{H}(n + i) = n_i : \tau_i \wedge n_i \neq 0))}{(\varphi, \mathcal{H}, \text{strlen}(n : \text{ptr}^m [(n_l, n_h) \tau])) \longrightarrow (\varphi, \mathcal{H}, n_a : \text{int})} \\
\\
\text{S-STRBOUNDS} \\
\frac{0 \notin [n_l, n_h]}{(\varphi, \mathcal{H}, \text{strlen}(n : \text{ptr}^c [(n_l, n_h) \tau])) \longrightarrow (\varphi, \mathcal{H}, \text{bounds})} \\
\\
\text{S-STRNULL} \\
(\varphi, \mathcal{H}, \text{strlen}(0 : \text{ptr}^c [(n_l, n_h) \tau])) \longrightarrow (\varphi, \mathcal{H}, \text{null}) \\
\\
\text{S-ADD} \\
\frac{n = n_1 + n_2}{(\varphi, \mathcal{H}, n_1 : \text{int} + n_2 : \text{int}) \longrightarrow (\varphi, \mathcal{H}, n)} \\
\\
\text{S-ADDARR} \\
\frac{n = n_1 + n_2 \quad n'_l = n_l - n_2 \quad n'_h = n_h - n_2}{(\varphi, \mathcal{H}, n_1 : \text{ptr}^m [(n_l, n_h) \tau]_{\kappa} + n_2 : \text{int}) \longrightarrow (\varphi, \mathcal{H}, n : \text{ptr}^m [(n'_l, n'_h) \tau]_{\kappa})} \\
\\
\text{S-ADDARRNULL} \\
n(\varphi, \mathcal{H}, 0 : \text{ptr}^c [(n_l, n_h) \tau]_{\kappa} + n_2 : \text{int}) \longrightarrow (\varphi, \mathcal{H}, \text{null})
\end{array}$$

Figure 15: Remaining CORECHKCBOX Semantics Rules (extends Fig. 7)

for array/NT-array pointers with pointer arithmetic. Rule T-UNCHECKED type checks **unchecked** blocks.

6. Struct Pointers

Checked C has struct types and struct pointers. Fig. 17 contains the syntax of struct types as well as new subtyping relations built on the struct values. For a struct typed value, Checked C has a special operation for it, which is $\&e \rightarrow f$. This operation indexes the f -th position struct T item, if the expression e is evaluated to a struct pointer ptr^m struct T . Rule T-STRUCT in Fig. 17 describes its typing behavior. Rules S-STRUCTCHECKED and S-STRUCTUNCHECKED describe the semantic behaviors of $\&e \rightarrow f$ on a given struct **checked/unchecked** pointers, while rule S-STRUCTNULL describes a **checked** struct null-pointer case. In our Coq/Redex formalization, we include the struct values and the operation $\&e \rightarrow f$. We omit it in the main text due to the paper length limitation.

7. The Compilation Rules

Fig. 22 and Fig. 23 shows the syntax for COREC, the target language for compilation. We syntactically restrict the expressions to be in A-normal form to simplify the presentation of the compilation rules. In the Redex model, we occasionally break this constraint to speed up the performance of random testing by removing unnecessary let bindings. To allow explicit runtime checks, we include bounds and null as part of COREC expressions which, once evaluated, result in an corresponding error state. $x = \dot{a}$ is a new syntactic form that modifies the stack variable x with the result of \dot{a} . It is essential for bounds widening. \leq and $-$ are introduced to operate on bounds and decide whether we should to halt with a bounds error or widen a null-terminated string.

COREC does not include any annotations. We remove structs from COREC because we can always statically convert expressions of the form $\&n : \tau \rightarrow f$ into $n + n_f$, where n_f is the statically determined offset of f within the struct.

$$\begin{array}{c}
\text{T-DEF} \\
\frac{\Gamma; \Theta \vdash_m e : \text{ptr}^{m'} \tau \quad m \leq m'}{\Gamma; \Theta \vdash_m * e : \tau} \\
\\
\text{T-MAC} \\
\Gamma; \Theta \vdash_m \text{malloc}(\omega, :) \text{ptr}^c \omega \\
\\
\text{T-ADD} \\
\frac{\Gamma; \Theta \vdash_m e_1 : \text{int} \quad \Gamma; \Theta \vdash_m e_2 : \text{int}}{\Gamma; \Theta \vdash_m (e_1 + e_2) : \text{int}} \\
\\
\text{T-IND} \\
\frac{\Gamma; \Theta \vdash_m e_1 : \text{ptr}^{m'} [\beta \tau]_\kappa \quad \Gamma; \Theta \vdash_m e_2 : \text{int} \quad m \leq m'}{\Gamma; \Theta \vdash_m * (e_1 + e_2) : \tau} \\
\\
\text{T-ASSIGN} \\
\frac{\Gamma; \Theta \vdash_m e_1 : \text{ptr}^{m'} \tau \quad \Gamma; \Theta \vdash_m e_2 : \tau' \quad \tau' \sqsubseteq \tau \quad m \leq m'}{\Gamma; \Theta \vdash_m * e_1 = e_2 : \tau} \\
\\
\text{T-INDASSIGN} \\
\frac{\Gamma; \Theta \vdash_m e_1 : \text{ptr}^{m'} [\beta \tau]_\kappa \quad \Gamma; \Theta \vdash_m e_2 : \text{int} \quad \Gamma; \Theta \vdash_m e_3 : \tau' \quad \tau' \sqsubseteq \tau \quad m \leq m'}{\Gamma; \sigma \vdash_m * (e_1 + e_2) = e_3 : \tau}
\end{array}$$

Figure 16: Remaining CORECHKCBOX Type Rules (extends Fig. 9)

We ellide the semantics of COREC because it is self-evident and mirrors the semantics CORECHKCBOX. The difference is that in COREC, only bounds and null can step into an error state. All failed dereferences and assignments would result in a stuck state and therefore we rely on the compiler to explicitly insert checks for checked pointers.

Fig. 26 and Fig. 27 shows the rules for the compilation judgment for expressions,

$$\Gamma; \rho \vdash e \gg \dot{C}, \dot{a}$$

The judgment is presented differently from the one in Sec. 4, which was simplified for presentation purposes. First, we remove Θ and m because these parameters are only used for checking and have no impact on compilation. Second, the judgment includes two outputs, a closure \dot{C} and an atom expression \dot{a} , instead of a single COREC expression \dot{e} . \dot{C} can be intuitively understood as a partially constructed program or context. Whereas \dot{E} is used for evaluation, \dot{C} is used purely as a device for compilation. As an example, when compiling $(1 : \text{int}) + (2 : \text{int})$, we would first create a fresh variable x , and then produce two outputs:

$$\begin{aligned}
\dot{C} &= \text{let } x = 1 + 2 \text{ in } \square \\
\dot{a} &= x
\end{aligned}$$

To obtain the compiled expression \dot{e} , we plug \dot{a} into \dot{C} using the usual notation $\dot{C}[\dot{a}]$. We can also use \dot{C} to represent runtime checks, which usually take the form $\text{let } x = \dot{c} \text{ in } \square$, where \dot{c} contains the check whose evaluation must not

Struct Syntax:

$$\begin{array}{ll}
\text{Type} & \text{struct } T \\
\text{Structdefs} & D \in T \rightarrow fs \\
\text{Fields} & fs ::= \tau \text{ f} \mid \tau \text{ f}; fs
\end{array}$$

Struct Subtype:

$$\begin{aligned}
D(T) = fs \wedge fs(0) = \text{nat} &\Rightarrow \text{ptr}^m \text{ struct } T \sqsubseteq \text{ptr}^m \text{ nat} \\
D(T) = fs \wedge fs(0) = \text{nat} \wedge 0 \leq b_l \wedge b_h \leq 1 &\Rightarrow \text{ptr}^m \text{ struct } T \sqsubseteq \text{ptr}^m [(b_l, b_h) \text{ nat}]
\end{aligned}$$

Struct Type Rule:

$$\frac{\text{T-STRUCT} \quad \Gamma; \Theta \vdash_m e : \text{ptr}^m \text{ struct } T \quad D(T) = fs \quad fs(f) = \tau_f}{\Gamma; \Theta \vdash_m \&e \rightarrow f : \text{ptr}^m \tau_f}$$

Struct Semantics:

$$\frac{\text{S-STRUCTCHECKED} \quad n > 0 \quad D(T) = fs \quad fs(f) = \tau_a \quad n_a = \text{index}(fs, f)}{(\varphi, \mathcal{H}, \&n : \text{ptr}^c \text{ struct } T \rightarrow f) \longrightarrow (\varphi, \mathcal{H}, n_a : \text{ptr}^c \tau_a)}$$

$$\frac{\text{S-STRUCTNULL} \quad n = 0}{(\varphi, \mathcal{H}, \&n : \text{ptr}^c \text{ struct } T \rightarrow f) \longrightarrow (\varphi, \mathcal{H}, \text{null})}$$

$$\frac{\text{S-STRUCTUNCHECKED} \quad D(T) = fs \quad fs(f) = \tau_a \quad n_a = \text{index}(fs, f)}{(\varphi, \mathcal{H}, \&n : \text{ptr}^u \text{ struct } T \rightarrow f) \longrightarrow (\varphi, \mathcal{H}, n_a : \text{ptr}^u \tau_a)}$$

Figure 17: CORECHKCBOX Struct Definitions

$$\begin{array}{lll}
\Gamma \vdash n & \frac{x : \text{int} \in \Gamma}{\Gamma \vdash x + n} & \frac{\Gamma \vdash b_l \quad \Gamma \vdash b_h}{\Gamma \vdash (b_l, b_h)} \quad \Gamma \vdash \text{int} \\
\\
\Gamma \vdash \beta & \Gamma \vdash \tau & \Gamma \vdash \tau \\
\Gamma \vdash \text{ptr}^m [\beta \tau]_\kappa & \Gamma \vdash \text{ptr}^m \tau & \frac{T \in D}{\Gamma \vdash \text{ptr}^m \text{ struct } T}
\end{array}$$

Figure 18: Well-formedness for Types and Bounds

trigger bounds or null for the program to continue (see Fig. 25 for the metafunctions that create those checks).

This unconventional output format enables us to separate the evaluation of the term and the computation that relies on the term's evaluated result. Since effects and reduction (except for variables) happen only within closures, we can precisely control the order in which effects and evaluation happen by composing the contexts in a specific order. Given two closures \dot{C}_1 and \dot{C}_2 , we write $\dot{C}_1[\dot{C}_2]$ to denote the meta operation of plugging \dot{C}_2 into \dot{C}_1 . We also use $\dot{C}_{a;b;c}$ as a shorthand for $\dot{C}_a[\dot{C}_b[\dot{C}_c]]$. In the C-IND rule, we first evaluate the expressions that correspond to e_1 and e_2 through \dot{C}_1 and \dot{C}_2 , and then perform a null check and an addition through \dot{C}_n and \dot{C}_3 . Finally, we dereference the result through \dot{C}_4 before returning the pair \dot{C}_4, \dot{x}_4 , propagating the flexibility to the compilation rule that recursively calls C-IND.

$$\frac{\Gamma \vdash \bar{x} : \bar{\tau} \quad \Gamma[\bar{x} \mapsto \bar{\tau}] \vdash \tau \quad \Gamma[\bar{x} \mapsto \bar{\tau}]; \Theta \vdash_e e : \tau}{\Gamma \vdash \tau (\bar{x} : \bar{\tau}) e} \quad \Gamma \vdash .$$

$$\frac{\Gamma \vdash \tau \quad \Gamma[x \mapsto \tau] \vdash \bar{x} : \bar{\tau}}{\Gamma \vdash x : \tau, \bar{x} : \bar{\tau}}$$

Figure 19: Well-formedness for functions

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \mathbf{f}} \quad \frac{\Gamma \vdash \tau \quad \Gamma \vdash fs}{\Gamma \vdash \tau \mathbf{f}; fs}$$

Figure 20: Well-formedness for structs

$$\frac{\Gamma[\bar{x} \mapsto \bar{\tau}]; \emptyset \vdash e \gg \dot{e} : \tau}{\Gamma \vdash \tau (\bar{x} : \bar{\tau}) e \gg (\bar{x}) \dot{e}}$$

Figure 21: Compilation Rules for Functions

Fig. 25 shows the metafunctions that create closures representing dynamic checks. These functions first examine whether the pointer is a checked. If the pointer is unchecked, an empty closure \square will be returned, because there is no need to perform a check. For bounds checking, there is a special case for NT-array pointers, where the bounds are retrieved from the **shadow** variables (found by looking up ρ) on the stack rather than using the bounds specified in the type annotation. This is how we achieve the same precise runtime behavior as CORECHKCBOX in our compiled expressions.

Fig. 24 shows the metafunctions related to bounds widening. \vdash_{extend} takes ρ , a checked NT-array pointer variable x , and its bounds (b_l, b_h) as inputs, and returns an extended ρ' that maps x to two fresh variables x_l, x_h , together with a closure \dot{C} that initializes x_l and x_h to b_l and b_h respectively. This function is used in the C-LET rule to extend ρ before compiling the body of the let binding. The updated ρ' can be used for generating precise bounds checks, and for inserting expressions that can potentially widen the upper bounds, as seen in the $\vdash_{widenstr}$ metafunction used in the C-STR compilation rule.

Atoms	$\dot{a} ::= n \mid x$
C-Expressions	$\dot{c} ::= \dot{a} \mid \mathbf{strlen}(\dot{a}) \mid \mathbf{malloc}(\dot{a}, \mid) f(\bar{\dot{a}})$ $\mid \dot{a} \circ \dot{a} \mid * \dot{a}$ $\mid * \dot{a} = \dot{a} \mid x = \dot{a} \mid \mathbf{if} (\dot{a}) \dot{e} \mathbf{else} \dot{e}$ $\mid \mathbf{bounds} \mid \mathbf{null}$
Expressions	$\dot{e} ::= \dot{c} \mid \mathbf{let} x = \dot{c} \mathbf{in} \dot{e}$
Binops	$\circ ::= + \mid - \mid \leq$
Closure	$\dot{C} ::= \square \mid \mathbf{let} x = \dot{a} \mathbf{in} \dot{C}$
Bounds Map	$\rho \in \mathbf{Var} \rightarrow \mathbf{Var} \times \mathbf{Var}$ $\mid \mathbf{if} (\dot{a}) \dot{e} \mathbf{else} \dot{C} \mid \mathbf{if} (\dot{a}) \dot{C} \mathbf{else} \dot{e}$

Figure 22: COREC Syntax

$$\begin{aligned}
\dot{\mu} &::= n \mid \perp \\
\dot{c} &::= \dots \mid \mathbf{ret}(x, \dot{\mu}, \dot{e}) \\
\dot{H} &\in \mathbb{Z} \rightarrow \mathbb{Z} \\
\dot{r} &::= \dot{e} \mid \mathbf{null} \mid \mathbf{bounds} \\
\dot{E} &::= \square \mid \mathbf{let} x = \dot{E} \mathbf{in} \dot{e} \mid \mathbf{ret}(x, i, \dot{E}) \\
&\mid \mathbf{if} (\dot{E}) \dot{e} \mathbf{else} \dot{e} \mid \mathbf{strlen}(\dot{E}) \\
&\mid \mathbf{malloc}(\dot{E}, \mid) f(\bar{\dot{E}}) \mid \dot{E} \circ \dot{a} \mid n \circ \dot{E} \\
&\mid * \dot{E} \mid * \dot{E} = \dot{a} \mid * n = \dot{E} \mid x = \dot{E} \\
\bar{\dot{E}} &::= \dot{E} \mid n, \bar{\dot{E}} \mid \bar{\dot{E}}, \dot{a}
\end{aligned}$$

Figure 23: COREC Semantic Defs

$$\begin{array}{c}
\frac{x_l, x_h = \mathbf{fresh} \quad \rho' = \rho[x \mapsto (x_l, x_h)] \quad \dot{C} = \mathbf{let } x_l = b_l \mathbf{ in let } x_h = b_h \mathbf{ in } \square}{\dot{C}, \rho' = \vdash_{\text{extend}} \rho, x, \mathbf{ptr}^c [(b_l, b_h) \tau]_{nt}} \\
\\
\frac{x_l, x_h = \rho(x) \quad x_w = \mathbf{fresh} \quad \dot{C} = \mathbf{let } x_w = \mathbf{if } (x_h) 0 \mathbf{ else } x_h = 1 \mathbf{ in } \square}{\dot{C} = \vdash_{\text{widenderef}} \rho, x, \mathbf{ptr}^c [(b_l, b_h) \tau]_{nt}} \quad \frac{e \notin \text{dom}(\rho)}{\square = \vdash_{\text{widenstr}} \rho, e, \dot{a}, \mathbf{ptr}^m [\beta \tau]_{nt}} \\
\\
\frac{x_l, x_h = \rho(e) \quad x_a = \mathbf{fresh} \quad \dot{C} = \mathbf{let } x_a = \mathbf{if } (\dot{a} \leq x_h) 0 \mathbf{ else } x_h = \dot{a} \mathbf{ in } \square}{\dot{C} = \vdash_{\text{widenstr}} \rho, e, \dot{a}, \mathbf{ptr}^c [\beta \tau]_{nt}}
\end{array}$$

Figure 24: Metafunctions for Widening

$$\begin{array}{c}
\frac{x = \mathbf{fresh} \quad \dot{C} = \mathbf{let } x = \mathbf{if } (\dot{a}) 0 \mathbf{ else null in } \square}{\dot{C} = \vdash_{\text{null}} \dot{a}, c} \quad \square = \vdash_{\text{null}} \dot{a}, u \\
\\
\square = \vdash_{\text{boundsR}} \rho, e, \mathbf{ptr}^u [\beta \tau]_{\kappa}, \dot{a} \\
\\
\frac{x_l, x_h = \rho(e) \quad \dot{C}_{cl} = \mathbf{let } x_{cl} = \mathbf{if } (x_l \leq \dot{a}) 0 \mathbf{ else bounds in } \square \quad \dot{C}_{ch} = \mathbf{let } x_{ch} = \mathbf{if } (\dot{a} \leq x_h) 0 \mathbf{ else bounds in } \square}{\dot{C}_{cl;ch} = \vdash_{\text{boundsR}} \rho, e, \mathbf{ptr}^c [\beta \tau]_{\kappa}, \dot{a}} \\
\\
\frac{e \notin \text{dom}(\rho) \quad x_l, x_h, x_{cl}, x_{ch} = \mathbf{fresh} \quad \dot{C}_l = \mathbf{let } x_l = b_l \mathbf{ in } \square \quad \dot{C}_h = \mathbf{let } x_h = b_h \mathbf{ in } \square \quad \dot{C}_{cl} = \mathbf{let } x_{cl} = \mathbf{if } (x_l \leq \dot{a}) 0 \mathbf{ else bounds in } \square \quad \dot{C}_{ch} = \mathbf{let } x_{ch} = \mathbf{if } (\dot{a} \leq x_h) 0 \mathbf{ else bounds in } \square}{\dot{C}_{l;h;cl;ch} = \vdash_{\text{boundsR}} \rho, e, \mathbf{ptr}^c [(b_l, b_h) \tau]_{nt}, \dot{a}} \\
\\
\frac{e \notin \text{dom}(\rho) \quad x_l, x_h, x_{cl}, x_{ch} = \mathbf{fresh} \quad \dot{C}_l = \mathbf{let } x_l = b_l \mathbf{ in } \square \quad \dot{C}_h = \mathbf{let } x_h = b_h \mathbf{ in } \square \quad \dot{C}_{cl} = \mathbf{let } x_{cl} = \mathbf{if } (x_l \leq \dot{a}) 0 \mathbf{ else bounds in } \square \quad \dot{C}_{ch} = \mathbf{let } x_{ch} = \mathbf{if } (x_h \leq \dot{a}) \mathbf{ bounds else } 0 \mathbf{ in } \square}{\dot{C}_{l;h;cl;ch} = \vdash_{\text{boundsR}} \rho, e, \mathbf{ptr}^c [(b_l, b_h) \tau]_{\kappa}, \dot{a}} \\
\\
\square = \vdash_{\text{boundsW}} \rho, e, \mathbf{ptr}^u [\beta \tau]_{\kappa}, \dot{a} \\
\\
\frac{x_l, x_h = \rho(e) \quad \dot{C}_{cl} = \mathbf{let } x_{cl} = \mathbf{if } (x_l \leq \dot{a}) 0 \mathbf{ else bounds in } \square \quad \dot{C}_{ch} = \mathbf{let } x_{ch} = \mathbf{if } (\dot{a} \leq x_h) 0 \mathbf{ else bounds in } \square}{\dot{C}_{cl;ch} = \vdash_{\text{boundsW}} \rho, e, \mathbf{ptr}^c [\beta \tau]_{\kappa}, \dot{a}} \\
\\
\frac{e \notin \text{dom}(\rho) \quad x_l, x_h, x_{cl}, x_{ch} = \mathbf{fresh} \quad \dot{C}_l = \mathbf{let } x_l = b_l \mathbf{ in } \square \quad \dot{C}_h = \mathbf{let } x_h = b_h \mathbf{ in } \square \quad \dot{C}_{cl} = \mathbf{let } x_{cl} = \mathbf{if } (x_l \leq \dot{a}) 0 \mathbf{ else bounds in } \square \quad \dot{C}_{ch} = \mathbf{let } x_{ch} = \mathbf{if } (x_h \leq \dot{a}) \mathbf{ bounds else } 0 \mathbf{ in } \square}{\dot{C}_{l;h;cl;ch} = \vdash_{\text{boundsW}} \rho, e, \mathbf{ptr}^c [(b_l, b_h) \tau]_{\kappa}, \dot{a}} \\
\\
\frac{e \notin \text{dom}(\rho) \quad x_l, x'_l, x_h, x'_h = \mathbf{fresh} \quad \dot{C}_1 = \mathbf{let } x_l = b_l \mathbf{ in let } x_h = b_h \mathbf{ in } \square \quad \dot{C}_2 = \mathbf{let } x'_l = b'_l \mathbf{ in let } x'_h = b'_h \mathbf{ in } \square \quad \dot{C}_3 = \mathbf{if } (x'_l \leq x_l) \square \mathbf{ else bounds} \quad \dot{C}_4 = \mathbf{if } (x_h \leq x'_h) \square \mathbf{ else bounds}}{\dot{C}_{1;2;3;4} = \vdash_{\text{boundsD}} \rho, e, \mathbf{ptr}^m [(b_l, b_h) \tau]_{\kappa}, \mathbf{ptr}^m [(b'_l, b'_h) \tau]_{\kappa}} \\
\\
\frac{x'_l, x'_h = \rho(e) \quad x_l, x_h = \mathbf{fresh} \quad \dot{C}_1 = \mathbf{let } x_l = b_l \mathbf{ in let } x_h = b_h \mathbf{ in } \square \quad \dot{C}_2 = \mathbf{if } (x'_l \leq x_l) \square \mathbf{ else bounds} \quad \dot{C}_3 = \mathbf{if } (x_h \leq x'_h) \square \mathbf{ else bounds}}{\dot{C}_{1;2;3} = \vdash_{\text{boundsD}} \rho, e, \mathbf{ptr}^m [(b_l, b_h) \tau]_{\kappa}, \mathbf{ptr}^m [(b'_l, b'_h) \tau]_{\kappa}}
\end{array}$$

Figure 25: Metafunctions for Dynamic Checks

$$\begin{array}{c}
\text{C-CONST} \\
\frac{}{\Gamma; \rho \vdash n : \tau \gg \square, n : \tau} \\
\\
\text{C-VAR} \\
\frac{x : \tau \in \Gamma}{\Gamma; \rho \vdash x \gg \square, x : \tau} \\
\\
\text{C-CAST} \\
\frac{\Gamma; \rho \vdash e \gg \dot{C}, \dot{a} : \tau'}{\Gamma; \rho \vdash (\tau)e \gg \dot{C}, \dot{a} : \tau} \\
\\
\text{C-DYNCAST} \\
\frac{\Gamma; \rho \vdash e \gg \dot{C}_1, \dot{a} : \mathbf{ptr}^m [\beta' \tau]_\kappa \quad \dot{C}_b = \vdash_{\text{bounds}D} \rho, e, \mathbf{ptr}^m [\beta \tau]_\kappa, \mathbf{ptr}^m [\beta' \tau]_\kappa}{\Gamma; \rho \vdash \langle \mathbf{ptr}^m [\beta \tau]_\kappa \rangle e \gg \dot{C}_{1;b}, \dot{a} : \mathbf{ptr}^m [\beta \tau]_\kappa} \\
\\
\text{C-STR} \\
\frac{\Gamma; \rho \vdash e \gg \dot{C}_1, \dot{a}_1 : \mathbf{ptr}^m [\beta \tau_a]_{nt} \quad \dot{C}_n = \vdash_{\text{null}} \dot{a}_1, m \quad \dot{C}_b = \vdash_{\text{bounds}R} \rho, \dot{a}_1, \mathbf{ptr}^m [\beta \tau_a]_{nt}, 0 \quad x_2 = \mathbf{fresh} \quad \dot{C}_2 = \mathbf{let } x_2 = \mathbf{strlen}(\dot{a}_1) \mathbf{in } \square \quad \dot{C}_w = \vdash_{\text{widenstr}} \rho, e, \dot{a}_1, \mathbf{ptr}^m [\beta \tau_a]_{nt}}{\Gamma; \rho \vdash \mathbf{strlen}(e) \gg \dot{C}_{1;n;b;2;w}, x_2 : \mathbf{int}} \\
\\
\text{C-LETSTR} \\
\frac{\Gamma(y) = \mathbf{ptr}^c [(b_l, b_h) \tau_a]_{nt} \quad x \notin FV(\tau) \quad \Gamma; \rho \vdash \mathbf{strlen}(y) \gg \dot{C}_1, \dot{a}_1 : \mathbf{int} \quad \dot{C}_2 = \mathbf{let } x = \dot{a}_1 \mathbf{in } \square \quad \Gamma[x \mapsto \mathbf{int}, y \mapsto [\mathbf{ptr}^c [(b_l, x) \tau_a]_{nt}]]; \rho \vdash e_3 \gg \dot{C}_3, \dot{a}_3 : \tau}{\Gamma; \rho \vdash \mathbf{let } x = \mathbf{strlen}(y) \mathbf{in } e \gg \dot{C}_{1;2;3}, \dot{a}_3 : \tau} \\
\\
\text{C-IF} \\
\frac{\Gamma; \rho \vdash e \gg \dot{C}_1, \dot{a}_1 : \tau \quad \Gamma; \rho \vdash e_1 \gg \dot{C}_2, \dot{a}_2 : \tau_2 \quad \Gamma; \rho \vdash e_3 \gg \dot{C}_3, \dot{a}_3 : \tau_3 \quad x_4 = \mathbf{fresh} \quad \dot{C}_4 = \mathbf{let } x_4 = \mathbf{if } (\dot{a}_1) \dot{C}_2[\dot{a}_2] \mathbf{else } \dot{C}_3[\dot{a}_3] \mathbf{in } \square}{\Gamma; \rho \vdash \mathbf{if } (e_1) e_2 \mathbf{else } e_3 \gg \dot{C}_{1;4}, x_4 : \tau_2 \sqcup \tau_3} \\
\\
\text{C-IFNT} \\
\frac{\Gamma; \rho \vdash x : \mathbf{ptr}^c [(b_l, b_h) \tau]_{nt} \quad b_h = 0 \Rightarrow \Gamma' = \Gamma[x \mapsto \mathbf{ptr}^c [(b_l, 1) \tau]_{nt}] \quad b_h \neq 0 \Rightarrow \Gamma' = \Gamma \quad \Gamma; \rho \vdash *x \gg \dot{C}_1, \dot{a}_1 : \tau_1 \quad \Gamma'; \rho \vdash e_2 \gg \dot{C}_2, \dot{a}_2 : \tau_2 \quad \Gamma; \rho \vdash e_3 \gg \dot{C}_3, \dot{a}_3 : \tau_3 \quad \dot{C}_w = \vdash_{\text{widenderef}} \rho, x, \mathbf{ptr}^c [(b_l, b_h) \tau]_{nt} \quad x_4 = \mathbf{fresh} \quad \dot{C}_4 = \mathbf{let } x_4 = \mathbf{if } (\dot{a}_1) \dot{C}_{2;w}[\dot{a}_2] \mathbf{else } \dot{C}_3[\dot{a}_3] \mathbf{in } \square}{\Gamma; \rho \vdash \mathbf{if } (*x) e_1 \mathbf{else } e_2 \gg \dot{C}_{1;4}, x_4 : \tau_1 \sqcup \tau_2} \\
\\
\text{C-LET} \\
\frac{(x \in FV(\tau') \Rightarrow e_1 \in \text{Bound}) \quad \Gamma; \rho \vdash e_1 \gg \dot{C}_1, \dot{a}_1 : \tau_1 \quad \dot{C}_2, \rho' = \vdash_{\text{extend}} \rho, x, \tau_1 \quad \dot{C}_3 = \mathbf{let } x = \dot{a}_1 \mathbf{in } \square \quad \Gamma[x \mapsto \tau]; \rho' \vdash e_4 \gg \dot{C}_4, \dot{a}_4 : \tau_4}{\Gamma; \rho' \vdash \mathbf{let } x = e_1 \mathbf{in } e_4 \gg \dot{C}_{1;2;3;4}, \dot{a}_4 : \tau_4[\tau_1 = \mathbf{int} \Rightarrow x \mapsto e_1]} \\
\\
\text{C-RET} \\
\frac{\Gamma(x) \neq \perp \quad \Gamma; \rho \vdash e \gg \dot{C}_1, \dot{a}_1 : \tau \quad x_2 = \mathbf{fresh} \quad \mu \gg \dot{\mu} \quad \dot{C}_2 = \mathbf{let } x_2 = \mathbf{ret}(x, \dot{\mu}, \dot{C}_1[\dot{a}_1]) \mathbf{in } \square}{\Gamma; \rho \vdash \mathbf{ret}(x, \mu, e) \gg \dot{C}_2, x_2 : \tau} \\
\\
\text{C-FUN} \\
\frac{\Xi(f) = \tau (\bar{x} : \bar{\tau}) e \quad (\forall e_i \in \bar{e} \quad \tau_i \in \bar{\tau} . \Gamma; \rho \vdash e_i \gg \dot{C}_i, \dot{a}_i : \tau'_i \wedge \tau'_i \sqsubseteq \tau_i[\bar{e}/\bar{x}]) \quad x_f = \mathbf{fresh} \quad \dot{C}_f = \mathbf{let } x_f = f(\bar{a}) \mathbf{in } \square}{\Gamma; \rho \vdash f(\bar{e}) \gg \dot{C}[\dot{C}_f], x_f : \tau[\bar{e}/\bar{x}]} \\
\\
\text{C-DEF} \\
\frac{\Gamma; \rho \vdash e_1 \gg \dot{C}_1, \dot{a}_1 : \mathbf{ptr}^m \tau \quad \dot{C}_n = \vdash_{\text{null}} \dot{a}_1, m \quad x_2 = \mathbf{fresh} \quad \dot{C}_2 = \mathbf{let } x_2 = * \dot{a}_1 \mathbf{in } \square}{\Gamma; \rho \vdash * e_1 \gg \dot{C}_{1;n;2}, x_2 : \tau} \\
\\
\text{C-DEFARR} \\
\frac{\Gamma; \rho \vdash e_1 \gg \dot{C}_1, \dot{a}_1 : \mathbf{ptr}^m [(b_l, b_h) \tau]_\kappa \quad \dot{C}_n = \vdash_{\text{null}} \dot{a}_1, m \quad \dot{C}_b = \vdash_{\text{bounds}R} \rho, e_1, \mathbf{ptr}^m [(b_l, b_h) \tau]_\kappa, 0 \quad x_2 = \mathbf{fresh} \quad \dot{C}_2 = \mathbf{let } x_2 = * \dot{a}_1 \mathbf{in } \square}{\Gamma; \rho \vdash * e_1 \gg \dot{C}_{1;n;b;2}, x_2 : \tau} \\
\\
\text{C-MAC} \\
\frac{\dot{C}_1, \dot{a}_1 = \mathbf{sizeof}(\omega) \quad x_2 = \mathbf{fresh} \quad \dot{C}_2 = \mathbf{let } x_2 = \mathbf{malloc}(\dot{a}_1,) \mathbf{in } \square}{\Gamma; \rho \vdash \mathbf{malloc}(\omega, \gg) \dot{C}_{1;2}, x_2 : \mathbf{ptr}^c \omega}
\end{array}$$

Figure 26: Compilation

$$\text{C-ADD} \quad \frac{\Gamma; \rho \vdash e_1 \gg \dot{C}_1, \dot{a}_1 : \text{int} \quad \Gamma; \rho \vdash e_2 \gg \dot{C}_2, \dot{a}_2 : \text{int} \quad x_3 = \text{fresh} \quad \dot{C}_3 = \text{let } x_3 = \dot{a}_1 + \dot{a}_2 \text{ in } \square}{\Gamma; \rho \vdash \dot{C}_3, x_3 : \text{int}}$$

$$\text{C-IND} \quad \frac{\Gamma; \rho \vdash e_1 \gg \dot{C}_1, \dot{a}_1 : \text{ptr}^m [\beta \tau]_\kappa \quad \Gamma; \rho \vdash e_2 \gg \dot{C}_2, \dot{a}_2 : \text{int} \quad \dot{C}_n = \vdash_{\text{null}} \dot{a}_1, m \quad \dot{C}_b = \vdash_{\text{boundsR}} \rho, e_1, \text{ptr}^m [\beta \tau]_\kappa, \dot{a}_2 \quad x_3, x_4 = \text{fresh} \quad \dot{C}_3 = \text{let } x_3 = \dot{a}_1 + \dot{a}_2 \text{ in } \square \quad \dot{C}_4 = \text{let } x_4 = * x_3 \text{ in } \square}{\Gamma; \rho \vdash * (e_1 + e_2) \gg \dot{C}_{1;2;n;3;b;4}, x_4 : \tau}$$

$$\text{C-ASSIGN} \quad \frac{\dot{C}_n = \vdash_{\text{null}} \dot{a}_1, m \quad \Gamma; \rho \vdash e_2 \gg \dot{C}_2, \dot{a}_2 : \tau' \quad \tau' \sqsubseteq \tau \quad x_3 = \text{fresh} \quad \dot{C}_3 = \text{let } x_3 = * \dot{a}_1 = \dot{a}_2 \text{ in } \square}{\Gamma; \rho \vdash * e_1 = e_2 \gg \dot{C}_{1;2;n;3}, x_3 : \tau}$$

$$\text{C-ASSIGNARR} \quad \frac{\Gamma; \rho \vdash e_1 \gg \dot{C}_1, \dot{a}_1 : \text{ptr}^{m'} [\beta \tau]_\kappa \quad \dot{C}_n = \vdash_{\text{null}} \dot{a}_1, m \quad \dot{C}_b = \vdash_{\text{boundsW}} \rho, e_1, \text{ptr}^m [(b_l, b_h) \tau]_\kappa, 0 \quad \Gamma; \rho \vdash e_2 \gg \dot{C}_2, \dot{a}_2 : \tau' \quad x_3 = \text{fresh} \quad \dot{C}_3 = \text{let } x_3 = * \dot{a}_1 = \dot{a}_2 \text{ in } \square \quad \tau' \sqsubseteq \tau}{\Gamma; \rho \vdash * e_1 = e_2 \gg \dot{C}_{1;2;n;b;3}, x_3 : \tau}$$

$$\text{C-INDASSIGN} \quad \frac{\Gamma; \rho \vdash e_1 \gg \dot{C}_1, \dot{a}_1 : \text{ptr}^m [\beta \tau]_\kappa \quad \Gamma; \rho \vdash e_2 \gg \dot{C}_2, \dot{a}_2 : \text{int} \quad \dot{C}_n = \vdash_{\text{null}} \dot{a}_1, m \quad \dot{C}_b = \vdash_{\text{boundsW}} \rho, e_1, \text{ptr}^m [\beta \tau]_\kappa, \dot{a}_2 \quad \Gamma; \rho \vdash e_3 \gg \dot{C}_3, \dot{a}_3 : \tau' \quad x_4, x_5 = \text{fresh} \quad \dot{C}_4 = \text{let } x_4 = \dot{a}_1 + \dot{a}_2 \text{ in } \square \quad \dot{C}_5 = \text{let } x_5 = * x_4 = x_3 \text{ in } \tau' \sqsubseteq \tau}{\Gamma; \rho \vdash * (e_1 + e_2) = e_3 \gg \dot{C}_{1;2;n;3;4;b;5} : \tau}$$

$$\text{C-STRUCT} \quad \frac{D(T) = \tau_0 \ f_0 \dots; \tau_j \ f; \dots \quad \Gamma; \rho \vdash e_1 \gg \dot{C}_1, \dot{a}_1 : \text{ptr}^m \text{ struct } T \quad \dot{C}_n = \vdash_{\text{null}} \dot{a}_1, m \quad x_2 = \text{fresh} \quad \dot{C}_2 = \text{let } x_2 = \dot{a}_1 + j \text{ in } \square}{\Gamma; \rho \vdash \&e_1 \rightarrow f \gg \dot{C}_2, x_2 : \text{ptr}^m \tau_f}$$

$$\text{C-UNCHECKED} \quad \frac{\Gamma; \rho \vdash e \gg \dot{C}, \dot{a} : \tau}{\Gamma; \rho \vdash \text{unchecked}(e) \{ \gg \} \dot{C}, \dot{a} : \tau}$$

Figure 27: Compilation (Continued)