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 [2, 25, 26] despite a large body of work that tries to prevent them [24]. Several industrial and research efforts—including CCured [20], Softbound [18], and ASAN [22]—have explored means to compile C programs to automatically enforce spatial safety. These approaches all impose performance overheads deemed too high for deployment use. Recently, Elliott et al. [5], Li et al. [14] 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 ported to Checked C [4], and overall, performance overhead seems low enough for practical deployment.

The guarantee Checked C provides focuses on checked core 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. But what are the guarantees of unchecked (unsafe) core regions? 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. Our first contribution is a clear program partition between checked and unchecked core regions, and ensures that a CHECKEDCBOX program execution *cannot crash* due to spatial safety violations. We restrict the checked and unchecked pointers usage only in checked and unchecked core regions, respectively; and develop tainted pointers for the communication between the two regions. Heap is partitioned into two parts: a *checked memory region* containing all checked pointers and an *unchecked memory region* that is sandboxed. Tainted pointers live in the unchecked memory region and is verified every time when using.

Formaling the Type System, Semantics and Compiler. Our second contribution is a core formalism named CORECHKCBOX, which extends Li et al. [14] 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 build a random generator of programs largely based on the typing rules of CORECHKCBOX and make sure that, both statically and dynamically, CORECHKCBOX and Clang Checked C are consistent after conversion. To the best of our knowledge, CORECHKCBOX is the first 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 core regions. In implementing a multi-threaded system, users might want to provide third party interface permitting third parties to write implement some new program features, while keeping those programs in unchecked core 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 overcome the discord, we develop two mechanisms in CHECKEDCBOX and maintain a stronger *non-exposure* guarantee on of the non-crashing guarantee; i.e., no checked pointer addresses can be observed in an unchecked region. The first mechanism is 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 regions possibly holding checked pointers. In this case, no checked pointer address can be observed in the unchecked code regions. In CHECKEDCBOX, for every checked function, we automatically compiler a tainted version by surrounding the function without a tainted shell.

We begin with a review of Checked C (Section 2) and introduce new features in CHECKEDCBOX, 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 discusses the key merge features between Checked-C and RLBox and the problems we are trying to solve.

Maintaining Non-crashing. Previously, the main guarantee of Checked C [14] was the blame theorem, i.e., if there is a crash, the source is from some unchecked regions. For example, at Figure 3 line 31, we call a unchecked function `f` with a checked null-terminated array (NT-array) pointer argument. 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.

The philosophy is that unchecked regions are written in C and unstable, and programmers will eventually remove them by converting them to Checked C code. The reality is that users might not want to convert everything. For some insignificant or handy history C code, such as library functions, they might want to keep it as long as no crashing.

The sources of crashing in Checked C are (1) unchecked regions crash themselves; and (2) the misuse of checked pointers in unchecked regions. Enlightened by RLBox [19], we sandbox the unchecked regions and then we utilize the Checked C type system to disallow checked pointers to be used in an unchecked region. To achieve the communication between checked and unchecked regions, we create tainted pointers that can be shared by different regions, whose data are stored in the sandboxed heap. Users are required to copy data to tainted pointers that are shared in unchecked regions. 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 handled by inserting additional checks and creating exception handling before the use by the CHECKEDCBOX compiler. Thus, the checked pointer `p` is safely used at line 38.

In short, banning checked pointer uses in unchecked regions and providing tainted shared pointers are the solution for crashing. In CHECKEDCBOX, we proved the non-crashing theorem based on the CHECKEDCBOX type system, .i.e, any program execution is stopped in a predictable manner where either the execution terminates or is stopped at a place where an exception handler exists. In fact, our work is *the first formalism of RLBox*, and formally shows the security guarantee that RLBox provides is actually the non-crashing theorem.

Formalism Function Pointers. In C, manipulating function pointers is a major way of implementing high order functions. Previously, we assumed all function calls are through calling by name to a global map. In CHECKEDCBOX, we formalize function pointers and maintain the CHECKEDCBOX type soundness. To the best of our knowledge, this is the first work of formalizing C function pointers with security guarantee.

Figure 1 defines a string sorting algorithm depending on the input function pointer `cmp` defining the generic order

```
//in checked region

int compare_1(nt_array_ptr<char> x: count (0),
  nt_array_ptr<char> y : count (0)) {
  int len_x = strlen(x);
  int len_y = strlen(y);
  return sum(x,len_x) < sum(y,len_y);
  }
...

int stringsort(
 nt_array_ptr<nt_array_ptr<char>> s : count (n),
 ptr<(int)(nt_array_ptr<char>,
  nt_array_ptr<char>)> cmp, int n) {
  int i, j, gap;
  int didswap;

  for(gap = n / 2; gap > 0; gap /= 2) {
    {
    do  {
        didswap = 0;
        for(i = 0; i < n - gap; i++)
           {
             j = i + gap;
             if((*cmp)(s[i], s[j]) > 0)
             {
                int len = strlen(s[i]);
                nt_array_ptr<char>
                  tmp : count (len) = s[i];
                s[i] = s[j];
                s[j] = tmp;
                didswap = 1;
             }
           }
        } while(didswap);
    }
  }

  return 0;
}
```

Figure 1: Checked stringsort Code

```
//in checked region, tainted version
int tainted_compare_1(
  nt_array_tptr<char> x : count (0),
  nt_array_tptr<char> y : count (0)) {
 checked (x,y) {
  int len_x = strlen(x);
  int len_y = strlen(y);
  nt_array_ptr<char> tx : count (len_x)
   = malloc(nt_array<char>, len_x);
  nt_array_ptr<char> ty : count (len_y)
   = malloc(nt_array<char>, len_y);
  safe_memcpy(tx,x,len_x);
  safe_memcpy(ty,y,len_y);
  return compare_1(tx,ty);
 }
}
...

//calling the function turns
//an unchecked region to a checked region.
int tainted_stringsort(nt_array_tptr
     <nt_array_tptr<char>> s : count (n),
 tptr<(int)(nt_array_tptr<char>,
  nt_array_tptr<char>)> cmp, int n) {
 checked (s,cmp,n) {
  int i;
  nt_array_ptr<nt_array_ptr<char>> p : count (n)
   = malloc(nt_array<nt_array_ptr<char>>, n);
  for(i = 0; i < n; i++) {
    int len = strlen s[i];
    nt_array_ptr<char> tmp : count (len)
      = new malloc(nt_array<char>, len);
    safe_memcpy(tmp,s[i],len);
    p[i] = tmp;
  }
  ptr<(int)(nt_array_ptr<char> : count (0),
  nt_array_ptr<char> : count (0))>
    cfun = find_check(cmp);

  return stringsort(p,cfun);
 }
}
```

Figure 2: Tainted stringsort Code

```
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,
            tainted_stringsort);
35   }
36
37   if (!BAD) safe_memcpy(p, tp, n);
38   p[10] = "crash?";
39
40   unchecked {
41     if (BAD) g(compare_1)
42       else g(tainted_compare_1);
43   }
44
45   return 0;
46 }
```

Figure 3: Tainted Pointer Usage in Calling Unchecked Fun

for strings, and `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. 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.

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 `compare_1`'s pointer 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-cast to the `cmp`'s type. In general, if function pointer x has type $*(tl \to t)$, and y has $*(tl' \to t')$, in order to use x as y, tl' should be a subtype of tl and t subtypes to t'.

Not Exposing Checked Pointer Addresses. The design for the non-crashing property bans the checked pointer uses in unchecked 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 mechanism, such as derandomizing ASLR [23], to achieve the checked pointer address. Thus, it enables a third party to access any checked heap and function data by simple pointer arithmetic.

To prevent the checked pointer address leak, we prevent any unchecked regions acknowledge checked pointer variables. 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`. In the tainted shells, the arguments are tainted versions of the corresponding arguments in the checked functions. Inside the shell body, create checked pointer copies of the tainted arguments, and call the checked functions. In addition, once a checked function returns, if the output is a checked pointer, we also its the data to a new tainted pointer and exit the shell. Figure 3 line 42 is an example of serving the function call living in an unchecked region with a tainted shell pointer argument `tainted_compare_1`. Even if g derndomizes its address (line 14), the shell address is in the sandbox and has no harm, and calling the shell never exposes any checked pointer information outside of the shell.

Conceptually, the shell is run in a checked region. One can imagine that a tainted shell is a safe closure that contains a checked block. Once the closure is called, the system is turned to a checked region, so that 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 in these tainted shell contains any checked pointers, as well as no output is of a checked type.

3. Formalization

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.

There are two type notions in CORECHKCBOX. Types τ classify word-sized values including integers and pointers,

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

Figure 4: CORECHKCBOX Syntax

$$
m \vdash \mathtt{int} \qquad \frac{\xi \wedge m \vdash \tau \qquad \xi \leq m}{m \vdash \mathtt{ptr}^{\xi}\ [\beta\ \tau]_{\kappa}}
$$

$$
\frac{\xi \wedge m \vdash \tau \qquad \xi \leq m}{m \vdash \mathtt{ptr}^{\xi}\ \tau} \qquad \frac{\xi \wedge m \vdash \tau \qquad \xi \leq m \qquad FV(\overline{\tau}) \cup FV(\tau) \subseteq \overline{x}}{m \vdash \mathtt{ptr}^{\xi}\ (\forall\, \overline{x}.\, \overline{\tau} \to \tau)}
$$

$$
\mathtt{t} \wedge \mathtt{c} = \mathtt{u} \qquad \xi \wedge \mathtt{u} = \mathtt{u} \qquad \mathtt{c} \wedge m = m \qquad m_1 \wedge m_2 = m_2 \wedge m_1
$$
$$
\xi \leq \xi \qquad \mathtt{t} \leq \xi
$$

Figure 5: Well-formedness for Types

while types ω classify multi-word values such as arrays, null-terminated arrays, functions, and single-word-size values. Pointer types ($\mathtt{ptr}^{\xi}\ \omega$) include a pointer mode annotation (ξ, the difference between context and pointer modes is introduced shortly below) that is either checked ($\mathtt{c}$), tainted ($\mathtt{t}$), or unchecked ($\mathtt{u}$), and a type (ω) denoting valid values that can be pointed to. 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). CORECHKCBOX function types ($\forall\, \overline{x}.\, \overline{\tau} \to \tau$) reflect its dependent function declarations, where $\overline{x}$ represents a list of `int`type variables in a dependent function header that bind bound variables appearing in $\overline{\tau}$ and τ. We have a well-formed requirement for a function type, such that all variables in $\overline{\tau}$ and τ are bounded by $\overline{x}$. Here is the corresponding CHECKEDCBOX syntax for these types:

$$
\begin{array}{rcl}
\mathtt{array\_tptr}\texttt{<}\tau\texttt{>}\ :\ \mathtt{count}(n) & \Leftrightarrow & \mathtt{ptr}^{\mathtt{t}}\ [(0, n)\ \tau] \\
\mathtt{nt\_array\_ptr}\texttt{<}\tau\texttt{>}\ :\ \mathtt{count}(n) & \Leftrightarrow & \mathtt{ptr}^{\mathtt{c}}\ [(0, n)\ \tau]_{nt}
\end{array}
$$
$$
\begin{array}{l}
\mathtt{tptr}\texttt{<(}\mathtt{int}\texttt{)(}\mathtt{nt\_array\_tptr}\texttt{<}\tau\texttt{>}\ :\ \mathtt{count}(n), \\
\qquad \mathtt{nt\_array\_tptr}\texttt{<}\tau\texttt{>)>}:\ \mathtt{count}(n)\texttt{))>} \\
\Leftrightarrow\ \mathtt{ptr}^{\mathtt{t}}\ (\forall\, n.\, \mathtt{ptr}^{\mathtt{t}}\ [(0, n)\ \tau]_{nt} \times \mathtt{ptr}^{\mathtt{t}}\ [(0, n)\ \tau]_{nt} \to \mathtt{int})
\end{array}
$$

As a convention we write $\texttt{ptr}^c\ [b\ \tau]$ to mean $\texttt{ptr}^c\ [(0,b)\ \tau]$, so the above examples could be rewritten $\texttt{ptr}^c\ [n\ \tau]$ and $\texttt{ptr}^c\ [n\ \tau]_{nt}$, respectively.

COREChkCBox expressions include literals $(n : \tau)$, variables (x), addition $(e_1 + e_2)$, static casts $((\tau)e)$, dynamic casts $(\langle\tau\rangle e)$ [1], the strlen operation $(\texttt{strlen}(x))$, pointer dereference and assignment $(*\ e)$ and $(*\ e_1 = e_2)$, resp.), let binding $(\texttt{let}\ x = e_1\ \texttt{in}\ e_2)$, conditionals $(\texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2)$, memory allocation $(\texttt{malloc}(\xi, \omega))$, function calls $(e(\overline{e}))$, unchecked blocks $(\texttt{unchecked}(\overline{x})\{e\})$, and checked blocks $(\texttt{checked}(\overline{x})\{e\})$.

Integer literals n are annotated with a type τ which can be either int, or $\texttt{ptr}^\xi\ \omega$ in the case n is being used as a heap address (this is useful for the semantics); $0 : \texttt{ptr}^\xi\ \omega$ (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

$\texttt{dyn\_bounds\_cast<array\_ptr<}\tau\texttt{>>(}e\texttt{, count(}n\texttt{))}$

becomes $\langle \texttt{ptr}^c\ [n\ \tau]\rangle e$.

Compared to the former Checked C model [14], 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 $\texttt{ptr}^\xi\ (...\texttt{ptr}^{\xi'}\ \tau...)$, $\xi' \leq \xi$. It is worth noting that pointer modes are a three point partial order $(\leq)$, where $\texttt{t}$ is the infimum, and $\xi \wedge m$ is a special meet operation that projects pointer modes onto context modes, such that $\texttt{t}$ is projected as $\texttt{u}$. Second, $\texttt{malloc}(\xi, \omega)$ includes a mode flag ξ for allocating different pointers in different heap mode regions. We disallow ω to be a function type $(\forall\ \overline{x}.\ \overline{\tau} \to \tau)$. Third, the first expression e in a function call $(e(\overline{e}))$ represents a function pointer. Fourth, checked blocks are added to the system, which permits the nested switching between checked (represented by context mode $\texttt{c}$) and unchecked (represented by context mode $\texttt{u}$) code regions. One example usage of the nested 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 $\texttt{checked}(\overline{x})\{e\}$ and $\texttt{unchecked}(\overline{x})\{e\}$: $\overline{x}$ 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 CheckedCBox 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.

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

$$
\begin{array}{lcl}
e & ::= & \ldots \mid \texttt{ret}(x, n{:}\tau, e) \\
r & ::= & e \mid \texttt{null} \mid \texttt{bounds} \\
E & ::= & \square \mid E + e \mid n{:}\tau + E \mid (\tau)E \mid \langle\tau\rangle E \mid *\ E \mid *\ E = e \\
 & & \mid *\ n{:}\tau = E \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{if}\ (E)\ e\ \texttt{else}\ e \\
 & & \mid E(\overline{e}) \mid n{:}\tau(\overline{E}) \mid \texttt{unchecked}(\overline{x})\{E\} \mid \texttt{checked}(\overline{x})\{E\}
\end{array}
$$

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

$$
\frac{\texttt{u} = \mathit{mode}(E) \qquad e = E[e'] \qquad \tau = \mathit{type}(e')}{(\varphi, \mathcal{H}, e) \longrightarrow_{\texttt{u}} (\varphi, \mathcal{H}, E[0{:}\tau])}
$$

$$
\begin{array}{l}
\mathit{mode}(E) = \mathit{mode}'(E, \texttt{c}) \\
\mathit{mode}'(\square, m) = m \\
\mathit{mode}'(\texttt{unchecked}(\overline{x})\{E\}, m) = \mathit{mode}'(E, \texttt{u}) \\
\mathit{mode}'(\texttt{checked}(\overline{x})\{E\}, m) = \mathit{mode}'(E, \texttt{c}) \\
\mathit{mode}'(\alpha(E), m) = \mathit{mode}'(E, m) \quad [\mathit{owise}]
\end{array}
$$

Figure 6: COREChkCBox Semantics: Evaluation

3.2. Semantics

The operational semantics for COREChkCBox is defined as a small-step transition relation with the judgment $(\varphi, \mathcal{H}, e) \longrightarrow_m (\varphi', \mathcal{H}', r)$. Here, φ is a *stack* mapping from variables to values $n : \tau$ and $\mathcal{H}$ is a *heap* that is partitioned into two parts ($\texttt{c}$ and $\texttt{u}$ regions), each of which maps addresses (integer literals) to values $n{:}\tau$. A cpointer is mapped to a heap location in the $\texttt{c}$ region, while a $\texttt{t}$ and $\texttt{u}$ pointer represents a $\texttt{u}$ region location. We wrote $\mathcal{H}(m, n)$ to retrieve the n-location heap value in the m region, and $\mathcal{H}(m)[n \mapsto n'{:}\tau]$ to update location n with the value $n'{:}\tau$ in the m region. 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. 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, representing 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) characterize undefined behavior. The context mode m (in $\longrightarrow_m$) indicates whether the stepped redex within e was in a checked ($\texttt{c}$) or unchecked ($\texttt{u}$) 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 region. 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. Evaluation contexts E define a standard left-to-right evaluation order. (We explain the $\texttt{ret}(x, \mu, e)$ syntax shortly.) There are other rules for describing the halts of evaluation to `null` and `bounds` states in Appendix A.

The *mode* function at the bottom of Fig. 6 describes the context mode determination in each evaluation step based on the context E. For any program execution, the function starts the mode computation with c ($mode(E) = mode'(E, \texttt{c})$). The result context mode depends on where $\square$ locates. If it occurs within E in $(\texttt{unchecked}(\overline{x})\{E\})$ that has no surrounding `checked` block, the mode is u; otherwise, the mode is c. $mode'(\alpha(E), m) = mode'(E, m)$ 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.

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 deference and assignment operations—illustrate how the semantics checks bounds. Rule S-DEFNULL transitions attempted null-pointer dereferences to `null`, whereas S-DEFC dereferences a c-mode non-null (single) pointer. When `null` 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 use of a tainted pointer requires a verification. Performing such a verification equates to performing a type check for a pointer constant in Figure 11. We explain this shortly in Section 3.3. For now, the verification step, e.g. $\emptyset; \mathcal{H}; \emptyset \vdash_{\texttt{u}} n_a : \tau$ in S-DEFC, means we verify 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 $\texttt{ptr}^{\texttt{c}}\ [(0, x+3)\ \texttt{int}]$, then $\varphi(\tau)$ would produce $\texttt{ptr}^{\texttt{c}}\ [(0,5)\ \texttt{int}]$ 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.

Binding and Function Calls. The semantics handles variable scopes using the special `ret` form. S-LET evaluates to a configuration whose expression is $\texttt{ret}(x, n{:}\tau, e))$. We keep φ unchanged and remember x and its the new value $n{:}\tau$ in e's scope that is defined by the `ret` operation. 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 $\texttt{ret}(x, \varphi'(x), e'))$, and restore x's outer score value $\varphi(x)$ in φ'. This procedure continues until e' becomes a literal $n : \tau$, in which case S-RETEND 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, which maps function pointers to function data $\tau\ (\overline{x} : \overline{\tau})\ (\xi, e)$, where τ is the return type, $(\overline{x} : \overline{\tau})$ is the parameter list of variables and their types, ξ determines the mode of the function, and e is the function body. Similar to $\mathcal{H}$, the global function store Ξ is also partitioned into two parts (c and u regions), each of which maps addresses (integer literals) to the function data described above.

The CHECKEDCBOX 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 $\overline{x}$ to the actual arguments $\overline{n}$, but annotated with the parameter types $\overline{\tau}$ (this will be safe for type-correct programs). The function body e is wrapped in a static cast $(\tau[\overline{n}/\overline{x}])$ which is the function's return type but with any integer parameter variables $\overline{x}$ appearing in that type, as type bound variables, substituted with the call's actual arguments $\overline{n}$. 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. Note that the S-FUNC and S-FUNT rules replace the annotations $\overline{\tau_a}$ with $\overline{\tau}$ (after instantiation) from the function's signature. Using $\overline{\tau_a}$ when executing the body of the function has no impact on the soundness of CORECHKCBOX, but will violate Theorem 6, which we introduce in Sec. 4. Rule S-FUNT defines the tainted version of function call semantics. In such case, the verification process $\emptyset; \mathcal{H}; \emptyset \vdash_{\texttt{u}} n : \texttt{ptr}^{\texttt{t}}\ \tau$ 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 CORECHKCBOX type system. The typing judgment has the form $\Gamma; \Theta \vdash_m e : \tau$, which states

S-DefC
$$\frac{\mathcal{H}(\mathtt{c},n) = n_a{:}\tau_a}{(\varphi,\mathcal{H},*\ n{:}\mathtt{ptr}^{\mathtt{c}}\ \tau) \longrightarrow (\varphi,\mathcal{H},n_a{:}\tau)}$$

S-AssignArrC
$$\frac{\mathcal{H}(\mathtt{c},n) = n_a{:}\tau_a \qquad 0 \in [n_l,n_h)}{(\varphi,\mathcal{H},*\ n{:}\mathtt{ptr}^{\mathtt{c}}\ [(n_l,n_h)\ \tau]_\kappa = n_1{:}\tau_1) \longrightarrow (\varphi,\mathcal{H}(\mathtt{c})[n \mapsto n_1{:}\tau_a],n_1{:}\tau)}$$

S-DefT
$$\frac{\mathcal{H}(\mathtt{u},n) = n_a{:}\tau_a \qquad \emptyset;\mathcal{H};\emptyset \vdash_{\mathtt{u}} n_a{:}\tau}{(\varphi,\mathcal{H},*\ n{:}\mathtt{ptr}^{\mathtt{t}}\ \tau) \longrightarrow (\varphi,\mathcal{H},n_a{:}\tau)}$$

S-AssignArrT
$$\frac{\mathcal{H}(\mathtt{u},n) = n_a{:}\tau_a \qquad 0 \in [n_l,n_h) \qquad \emptyset;\mathcal{H};\emptyset \vdash_{\mathtt{u}} n_1{:}\tau}{(\varphi,\mathcal{H},*\ n{:}\mathtt{ptr}^{\mathtt{t}}\ [(n_l,n_h)\ \tau]_\kappa = n_1{:}\tau_1) \longrightarrow (\varphi,\mathcal{H}(\mathtt{u})[n \mapsto n_1{:}\tau_a],n_1{:}\tau)}$$

S-DefNull
$$(\varphi,\mathcal{H},*\ 0{:}\mathtt{ptr}^{\mathtt{c}}\ \omega) \longrightarrow (\varphi,\mathcal{H},\mathtt{null})$$

S-Cast
$$(\varphi,\mathcal{H},(\tau)n{:}\tau') \longrightarrow (\varphi,\mathcal{H},n{:}\varphi(\tau))$$

S-RetEnd
$$(\varphi,\mathcal{H},\mathtt{ret}(x,n{:}\tau,n'{:}\tau')) \longrightarrow (\varphi,\mathcal{H},n'{:}\tau')$$

S-Let
$$(\varphi,\mathcal{H},\mathtt{let}\ x = n{:}\tau\ \mathtt{in}\ e) \longrightarrow (\varphi,\mathcal{H},\mathtt{ret}(x,n{:}\tau,e))$$

S-RetCon
$$\frac{(\varphi[x \mapsto n{:}\tau],\mathcal{H},e) \longrightarrow (\varphi',\mathcal{H}',e')}{(\varphi,\mathcal{H},\mathtt{ret}(x,n{:}\tau,e)) \longrightarrow (\varphi'[x \mapsto \varphi(x)],\mathcal{H}',\mathtt{ret}(x,\varphi'(x),e'))}$$

S-Unchecked
$$(\varphi,\mathcal{H},\mathtt{unchecked}(\overline{x})\{n{:}\tau\}) \longrightarrow (\varphi,\mathcal{H},n{:}\tau)$$

S-FunC
$$\frac{\Xi(\mathtt{c},n) = \tau\ (\overline{x}{:}\overline{\tau})\ (\mathtt{c},e)}{(\varphi,\mathcal{H},n{:}(\mathtt{ptr}^{\mathtt{c}}\ \tau)(\overline{n_a{:}\tau_a})) \longrightarrow (\varphi,\mathcal{H},\mathtt{let}\ \overline{x} = \overline{n}{:}(\overline{\tau}[\overline{n}/\overline{x}])\ \mathtt{in}\ (\tau[\overline{n}/\overline{x}])e)}$$

S-Checked
$$(\varphi,\mathcal{H},\mathtt{checked}(\overline{x})\{n{:}\tau\}) \longrightarrow (\varphi,\mathcal{H},n{:}\tau)$$

S-FunT
$$\frac{\Xi(\mathtt{u},n) = \tau\ (\overline{x}{:}\overline{\tau})\ (\mathtt{t},e) \qquad \emptyset;\mathcal{H};\emptyset \vdash_{\mathtt{u}} n{:}\mathtt{ptr}^{\mathtt{t}}\ \tau}{(\varphi,\mathcal{H},n{:}(\mathtt{ptr}^{\mathtt{t}}\ \tau)(\overline{n_a{:}\tau_a})) \longrightarrow (\varphi,\mathcal{H},\mathtt{let}\ \overline{x} = \overline{n}{:}(\overline{\tau}[\overline{n}/\overline{x}])\ \mathtt{in}\ (\tau[\overline{n}/\overline{x}])e)}$$

Figure 7: CoreChkCBox Semantics: Computation (Selected Rules)

```
nt_array_ptr<char> safe_strcat
   (nt_array_ptr<char> dst : count(n),
    nt_array_ptr<char> src : count(0), int n) {
  int x = strlen(dst);
  int y = strlen(src);
  nt_array_ptr<char> c : count(n) =
    dyn_bounds_cast
          <nt_array_ptr<char>>(dst,count(n));
    // sets c == dst with bound n (not x)
  if (x+y < n) {
    for (int i = 0; i < y; ++i)
      *(c+x+i) = *(src+i);
    *(c+x+y) = '\0';
    return dst;
  }
  return null;
}
```

Figure 8: Implementation of safe strcat

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. Rules T-Def and T-AssignArr check array dereference and assignment operations resp., 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 deferenced in checked and unchecked regions, resp.; The type rules do not attempt to reason whether the access is in bounds; such check is deferred to the semantics.

Type Equality and Subtyping and Casting. In CoreChkCBox, 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. 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 $\forall\ \overline{x}.\ \overline{\tau} \to \tau$ and $\forall\ \overline{y}.\ \overline{\tau'} \to \tau'$ are equivalent, if we can find a same length, as $\overline{x}$ and $\overline{y}$, variable list $\overline{z}$ that is substituted for $\overline{x}$ and $\overline{y}$ in $\overline{\tau} \to \tau$ and $\overline{\tau'} \to \tau'$, resp., and the substitution results are equal.

The T-CastPtr rule permits casting from an expression of type τ' to a checked pointer when $\tau' \sqsubseteq \mathtt{ptr}^{\mathtt{c}}\ \tau$. This subtyping relation $\sqsubseteq$ is given in Fig. 10 and it is built on the type equality ($\tau =_\Theta \tau' \Rightarrow \tau \sqsubseteq_\Theta \tau'$); the many rules ensure the relation is transitive. Most of the rules handle casting between array pointer types. The second rule $0 \leq b_l \wedge b_h \leq 1 \Rightarrow \mathtt{ptr}^m\ \tau \sqsubseteq \mathtt{ptr}^m\ [(b_l,b_h)\ \tau]$ 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 ($\mathtt{ptr}^\xi\ \forall\,\overline{x}.\,\overline{\tau} \to \tau \sqsubseteq_\Theta \mathtt{ptr}^\xi\ \forall\,\overline{x}.\,\overline{\tau'} \to \tau'$), if the output type are subtyped ($\tau \sqsubseteq_\Theta \tau'$) and the argument types are reversely subtyped ($\overline{\tau'} \sqsubseteq_\Theta \overline{\tau}$). There is another casting rule in Appendix A states that users are free to cast types in unchecked code regions, since unchecked regions can contain C code.

Since bounds expressions may contain variables, determining assumptions like $b_l \leq b_l'$ requires reasoning about those variables' possible values. The type system uses Θ to make such reasoning more precise. Θ is a map from variables x to equation predicates P, which have the form $P ::= \mathtt{ge\_0} \mid \mathtt{eq}\ b$. It maps variables to equations that are

$$\textsc{T-ConstU}\quad \frac{\neg\mathtt{c}(\tau)}{\Gamma;\Theta\vdash_u n{:}\tau:\tau}
\qquad
\textsc{T-ConstC}\quad \frac{\Theta;\mathcal{H};\emptyset\vdash_c n:\tau}{\Gamma;\Theta\vdash_c n{:}\tau:\tau}
\qquad
\textsc{T-Def}\quad \frac{\begin{array}{c}\xi\le m\\ \Gamma;\Theta\vdash_m e:\mathtt{ptr}^{\xi}\ \tau\end{array}}{\Gamma;\Theta\vdash_m *\,e:\tau}$$

$$\textsc{T-AssignArr}\quad \frac{\begin{array}{c}\Gamma;\Theta\vdash_m e_1:\mathtt{ptr}^{\xi}\ [\beta\ \tau]_\kappa\\ \Gamma;\Theta\vdash_m e_2:\tau'\\ \tau'\sqsubseteq_\Theta\tau \qquad \xi\le m\end{array}}{\Gamma;\Theta\vdash_m *\,e_1=e_2:\tau}
\qquad
\textsc{T-CastPtr}\quad \frac{\Gamma;\Theta\vdash_m e:\tau' \qquad \tau'\sqsubseteq_\Theta \mathtt{ptr}^{\xi}\ \tau}{\Gamma;\Theta\vdash_m (\mathtt{ptr}^{\xi}\ \tau)e:\mathtt{ptr}^{\xi}\ \tau}$$

$$\textsc{T-Let}\quad \frac{\begin{array}{c}x\notin FV(\tau') \qquad \Gamma;\Theta\vdash_m e_1:\tau\\ \Gamma[x\mapsto\tau];\Theta\vdash_m e_2:\tau'\end{array}}{\Gamma;\Theta\vdash_m \mathtt{let}\ x=e_1\ \mathtt{in}\ e_2:\tau'}
\qquad
\textsc{T-LetInt}\quad \frac{\begin{array}{c}x\in FV(\tau')\Rightarrow e_1\in\text{Bound}\\ \Gamma;\Theta\vdash_m e_1:\mathtt{int}\\ \Gamma[x\mapsto\mathtt{int}];\Theta[x\mapsto\mathtt{eq}\ e_1]\vdash_m e_2:\tau'\end{array}}{\Gamma;\Theta\vdash_m \mathtt{let}\ x=e_1\ \mathtt{in}\ e_2:\tau'[e_1/x]}$$

$$\textsc{T-RetInt}\quad \frac{\Gamma[x\mapsto\mathtt{int}];\Theta[x\mapsto\mathtt{eq}\ n]\vdash_m e:\tau}{\Gamma;\Theta\vdash_m \mathtt{ret}(x,n{:}\mathtt{int},e):\tau}$$

$$\textsc{T-Checked}\quad \frac{\begin{array}{c}\forall x\in\overline{x}\,.\,\neg\mathtt{c}(\Gamma(x)) \qquad \neg\mathtt{c}(\tau)\\ FV(e)\in\overline{x} \qquad \Gamma;\Theta\vdash_c e:\tau\end{array}}{\Gamma;\Theta\vdash_m \mathtt{checked}(\overline{x})\{e\}:\tau}
\qquad
\textsc{T-Unchecked}\quad \frac{\begin{array}{c}\forall x\in\overline{x}\,.\,\neg\mathtt{c}(\Gamma(x)) \qquad \neg\mathtt{c}(\tau)\\ FV(e)\in\overline{x} \qquad \Gamma;\Theta\vdash_u e:\tau\end{array}}{\Gamma;\Theta\vdash_m \mathtt{unchecked}(\overline{x})\{e\}:\tau}$$

$$\textsc{T-Fun}\quad \frac{\begin{array}{c}\Gamma;\Theta\vdash_m e:\mathtt{ptr}^{\xi}\ \forall\,\overline{x}.\,\overline{\tau}\to\tau \qquad \Gamma;\Theta\vdash_m \overline{e}:\overline{\tau'}\\ \overline{e'}=\{e'|(e',\mathtt{int})\in(\overline{e}:\overline{\tau'})\}\\ \forall e'\,.\,e'\in\overline{e'}\Rightarrow e'\in\text{Bound} \qquad \overline{\tau'}\sqsubseteq_\Theta\overline{\tau}[\overline{e'}/\overline{x}]\end{array}}{\Gamma;\Theta\vdash_m e(\overline{e}):\tau[\overline{e'}/\overline{x}]}$$

$$\mathtt{c}(\mathtt{int})=\mathtt{false} \qquad \mathtt{c}(\mathtt{ptr}^{c}\ \omega)=\mathtt{true} \qquad \mathtt{c}(\mathtt{ptr}^{\xi}\ \omega)=\mathtt{false}\ \ [\mathit{owise}]$$

Figure 9: Selected type rules

Bound Inequality and Equality:

$$\begin{array}{rcl}
n\le n' \Rightarrow n & \le_\Theta & n'\\
n\le n' \Rightarrow x+n & \le_\Theta & x+n'\\
n\le n'\wedge\Theta(x)=\mathtt{ge_0} \Rightarrow n & \le_\Theta & x+n'\\
\Theta(x)=\mathtt{eq}\,b\wedge b+n\le_\Theta b' \Rightarrow x+n & \le_\Theta & b'\\
\Theta(x)=\mathtt{eq}\,b\wedge b'\le_\Theta b+n \Rightarrow b' & \le_\Theta & x+n\\
b\le_\Theta b'\wedge b'\le_\Theta b \Rightarrow b & =_\Theta & b'
\end{array}$$

Type Equility:

$$\begin{array}{rcl}
\mathtt{int} & =_\Theta & \mathtt{int}\\
\omega=_\Theta\omega' \Rightarrow \mathtt{ptr}^{\xi}\ \omega & =_\Theta & \mathtt{ptr}^{\xi}\ \omega'\\
\beta=_\Theta\beta'\wedge\tau=_\Theta\tau' \Rightarrow [\beta\ \tau]_\kappa & =_\Theta & [\beta'\ \tau']_\kappa\\
\mathit{cond}(\overline{x},\overline{\tau}\to\tau,\overline{y},\overline{\tau'}\to\tau') \Rightarrow \forall\,\overline{x}.\,\overline{\tau}\to\tau & =_\Theta & \forall\,\overline{y}.\,\overline{\tau'}\to\tau'
\end{array}$$

Subtype:

$$\begin{array}{rcl}
\tau=_\Theta\tau' \Rightarrow \tau & \sqsubseteq_\Theta & \tau'\\
0\le_\Theta b_l\wedge b_h\le_\Theta 1 \Rightarrow \mathtt{ptr}^{m}\ \tau & \sqsubseteq_\Theta & \mathtt{ptr}^{m}\ [(b_l,b_h)\ \tau]\\
b_l\le_\Theta 0\wedge 1\le_\Theta b_h \Rightarrow \mathtt{ptr}^{m}\ [(b_l,b_h)\ \tau] & \sqsubseteq_\Theta & \mathtt{ptr}^{m}\ \tau\\
b_l\le_\Theta 0\wedge 1\le_\Theta b_h \Rightarrow \mathtt{ptr}^{m}\ [(b_l,b_h)\ \tau]_{nt} & \sqsubseteq_\Theta & \mathtt{ptr}^{m}\ \tau\\
b_l\le_\Theta b_l'\wedge b_h'\le_\Theta b_h \Rightarrow \mathtt{ptr}^{m}\ [(b_l,b_h)\ \tau]_{nt} & \sqsubseteq_\Theta & \mathtt{ptr}^{m}\ [(b_l',b_h')\ \tau]\\
b_l\le_\Theta b_l'\wedge b_h'\le_\Theta b_h \Rightarrow \mathtt{ptr}^{m}\ [(b_l,b_h)\ \tau]_\kappa & \sqsubseteq_\Theta & \mathtt{ptr}^{m}\ [(b_l',b_h')\ \tau]_\kappa\\
\overline{\tau'}\sqsubseteq_\Theta\overline{\tau}\wedge\tau\sqsubseteq_\Theta\tau' \Rightarrow \mathtt{ptr}^{\xi}\ \forall\,\overline{x}.\,\overline{\tau}\to\tau & \sqsubseteq_\Theta & \mathtt{ptr}^{\xi}\ \forall\,\overline{x}.\,\overline{\tau'}\to\tau'
\end{array}$$

$$n'+n=\mathit{add}(n',n) \qquad (x+n')+n=x+\mathit{add}(n',n)$$

$$\begin{array}{l}\mathit{cond}(\overline{x},\tau,\overline{y},\tau')=\exists\overline{z}\,.\,\overline{x}\cup\overline{z}\wedge\overline{y}\cup\overline{z}\wedge\mathit{size}(\overline{x})=\mathit{size}(\overline{y})=\mathit{size}(\overline{z})\\ \qquad\qquad\wedge\tau[\overline{z}/\overline{x}]=\tau'[\overline{z}/\overline{x}]\end{array}$$

Figure 10: Type Equality and Subtyping

$$\Theta;\mathcal{H};\sigma\vdash_m n:\mathtt{int} \qquad \Theta;\mathcal{H};\sigma\vdash_m 0:\mathtt{ptr}^{\xi}\ \omega$$

$$\frac{\begin{array}{c}(m=\mathtt{c}\Rightarrow\xi\neq\mathtt{c})\\ (m=\mathtt{u}\Rightarrow\xi=\mathtt{u})\end{array}}{\Theta;\mathcal{H};\sigma\vdash_{\mathtt{c}} n:\mathtt{ptr}^{\mathtt{t}}\ \omega}
\qquad
\frac{(n{:}\mathtt{ptr}^{\xi}\ \omega)\in\sigma}{\Theta;\mathcal{H};\sigma\vdash_m n:\mathtt{ptr}^{\xi}\ \omega}$$

$$\frac{\mathtt{ptr}^{\xi'}\ \omega'\sqsubseteq_\Theta\mathtt{ptr}^{\xi}\ \omega \qquad \Theta;\mathcal{H};\sigma\vdash_m n:\mathtt{ptr}^{\xi'}\ \omega'}{\Theta;\mathcal{H};\sigma\vdash_m n:\mathtt{ptr}^{\xi}\ \omega}$$

$$\frac{\begin{array}{c}\xi\le m\\ \Xi(m,n)=\tau\ (\overline{x'}{:}\overline{\tau})\ (\xi,e) \qquad \overline{x}=\{x|(x:\mathtt{int})\in(\overline{x'}:\overline{\tau})\}\end{array}}{\Theta;\mathcal{H};\sigma\vdash_m n:\mathtt{ptr}^{\xi}\ (\forall\,\overline{x}.\,\overline{\tau}\to\tau)}$$

$$\frac{\begin{array}{c}\neg\mathit{fun\_t}(\omega) \qquad \xi\le m\\ \forall i\in[0,\mathit{size}(\omega))\,.\,\Theta;\mathcal{H};(\sigma\cup\{(n:\mathtt{ptr}^{\xi}\ \omega)\})\vdash_m \mathcal{H}(m,n+i)\end{array}}{\Theta;\mathcal{H};\sigma\vdash_m n:\mathtt{ptr}^{\xi}\ \omega}$$

$$\mathit{fun\_t}(\forall\,\overline{x}.\,\overline{\tau}\to\tau)=\mathtt{true} \qquad \mathit{fun\_t}(\omega)=\mathtt{false}\ \ [\mathit{owise}]$$

Figure 11: Verification/Type Rules for Constants

recorded along the type checking procedure. If Θ maps x to ge_0, that means that $x\ge 0$; $\mathtt{eq}\ b$ 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. $\mathtt{c}(\tau)$ judges 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 CheckedCBox. In rule T-ConstC, we requires a static verification procedure for validating a constant pointer, which is similar to the dynamic verification process in Section 3.2.

Given a constant $n:\tau$, the verification process $\Theta;\mathcal{H};\sigma\vdash_m n:\tau$ checks (Figure 11) if the constant is valid, 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 $\Xi(m)$ is also required to check the validity of a function pointer. A valid function pointer should appear in the right store region ($\mathtt{c}$ or $\mathtt{u}$) and the address stores a function with the right type. The last rule in Figure 11 describes the validity check for a non-function pointer, where every element in the pointer range ($[0, \mathit{size}(\omega))$) should be well typed.

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.

Unchecked and Checked Blocks. During the type checking, Both $\mathtt{checked}(\overline{x})\{e\}$ and $\mathtt{unchecked}(\overline{x})\{e\}$ check all free variables in e are within $\overline{x}$; the types for $\overline{x}$ and the final return type τ have no checked pointers. 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.

Let Bindings and Dependent Function Pointers. Rules T-LET and T-LETINT type a `let` expression, which also admits type dependency. 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.

Rule T-FUN is the dependent function call rule. Given a function pointer type ($\mathtt{ptr}^{\xi}\ \forall\, \overline{x}.\ \overline{\tau} \to \tau$) from a type-check for e and the types $\overline{\tau'}$ from the argument type checks for $\overline{e}$, we confirm that each of $\overline{\tau'}$ is a subtype of the corresponding one in $\overline{\tau}[\overline{e'}/\overline{x}]$, which replaces possible integer bound variables $\overline{x}$ with bound expressions $\overline{e'}$. The final result type is the defined target type τ appearing in the function pointer type also with such replacement, written as $\tau[\overline{e'}/\overline{x}]$. 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

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

Type soundness relies on several *well-formedness*:

Definition 1 (Type Environment Well-formedness). A type environment Γ is well-formed iff 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 iff (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 iff 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.

Definition 4 (Stack Consistency). A type environment Γ, variable predicate set Θ, and stack snapshot φ are consistent—written $\Gamma; \Theta \vdash \varphi$—iff 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 $\mathcal{H} \vdash \varphi$—iff for every variable x, $\varphi(x) = n\,{:}\,\tau$ with $\mathit{mode}(\tau) = \mathtt{c}$ implies $\emptyset; \mathcal{H}(\mathtt{c}); \emptyset \vdash_{\mathtt{c}} n : \tau$.

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

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

Theorem 1 (Progress).
For any COREСHKCBOX 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 ($\Gamma; \Theta \vdash_{\mathtt{c}} e : \tau$ for some τ), one of the following holds:

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

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 COREСHKCBOX 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 ($\Gamma; \Theta \vdash_{\mathtt{c}} e : \tau$ for some τ), if there exists φ', $\mathcal{H}'$ and e', such that $(\varphi, \mathcal{H}, e) \longrightarrow_{\mathtt{c}} (\varphi', \mathcal{H}', e')$, 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 ($\Gamma'; \Theta \vdash_{\mathtt{c}} e : \tau'$), where $\tau' \sqsubseteq_{\Theta} \tau$.

Theorem 3 (Unchecked Preservation). For any COREСHKCBOX program e, heap $\mathcal{H}$, stack φ,

	c		t		u	
	CBox / Core		CBox / Core		CBox / Core	
c	$*\,x$ / $*(\mathtt{c}, x)$		sand_get(x) / $*(\mathtt{u}, x)$		$\times$	
u	$\times$		$*\,x$ / $*(\mathtt{u}, x)$		$*\,x$ / $*(\mathtt{u}, x)$	

Figure 12: Compiled Targets for Dereference

type environment Γ, and variable predicate set Θ that are all are well-formed and well typed ($\Gamma; \Theta \vdash_{\mathtt{c}} e : \tau$ for some τ), if there exists φ', $\mathcal{H}'$ and e', such that $(\varphi, \mathcal{H}, e) \longrightarrow_{\mathtt{u}} (\varphi', \mathcal{H}', e')$, then $\mathcal{H}'(\mathtt{c}) = \mathcal{H}(\mathtt{c})$.

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 ($\Gamma; \Theta \vdash_{\mathtt{c}} e : \tau$ for some τ), if there exists φ', $\mathcal{H}'$ and e', such that $(\varphi, \mathcal{H}, e) \longrightarrow_{\mathtt{u}} (\varphi', \mathcal{H}', e')$ and $e = E[\alpha(x)]$ and *mode*$(E) = \mathtt{u}$, 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 step[2]).

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 ($\varphi; \Theta \vdash_{\mathtt{c}} e : \tau$ for some τ) and there exists φ_i, $\mathcal{H}_i$, e_i, and m_i for $i \in [1, k]$, such that $(\varphi, \mathcal{H}, e) \longrightarrow_{m_1} (\varphi_1, \mathcal{H}_1, e_1) \longrightarrow_{m_2} \ldots \longrightarrow_{m_k} (\varphi_k, \mathcal{H}_k, r)$, then r can never be *stuck*.

4. Compilation

The main subtlety of compiling Checked C to Clang/LLVM 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 [14] inserted additional pointer checks for verifying pointers being not null and the bounds are within their 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. 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 syntax when the modes are different. Figure 12 lists the different compiled syntax for a deference 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 deference operation to the sandbox pointer access function (sand_get(x)) accessing the data in the CheckedCBox implementation. In CoreChkCBox, we create a new deference 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 [14] 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 deference, assignment, malloc, function calls are: $*(m, e)$, $*(m, e) = e$, malloc(m, ω), and $(m, e)(\overline{e})$. The algorithm sheds light on how compilation can be implemented in the real Checked C compiler, while eschewing many important details (CoreC has many differences with LLVM IR).

Compilation is defined by extending CoreChkCBox's typing judgment thusly:

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

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.[3]

We formalize rules for this judgment in PLT Redex [7], 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 example, 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 handled by storing results of subexpressions into temporary variables, as in the following example.

2. 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.

3. 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 : ptr^c [(0,n) int]_nt,
3       q : ptr^t [(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 : ptr^c [(0,5) int]_nt */
11 /* q0 : ptr^t [(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

```
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 (or the beginning of a function body) of a pointer, if the poniter is an (NT-)array one, we first allocates two *shadow variables* to track the lower and upper bounds potentially changed for pointer arithmetic and NT-array bound widening. We also associated with every `c`-mode pointer variable according to its type; 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 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. In implementation, we actually optimize the verification away and substitute it with the bounds and null-pointer checks. Because tainted pointer is checked every time before it is used, so that we only need to check the top pointer is well defined without recursively looking at the sub-terms in a nested pointer case. The modes in compiled deference (`*(mode(p)∧m,p)`) and assignment (`*(mode(q)∧m,q)=1`) operations are computed based on the meet operation (∧) of the pointer mode (e.g. `mode(p)`) and the current context mode (`m`).

Checked and Unchecked Blocks. In the CHECKEDCBOX implementation, `unchecked` and `checked` blocks are compiled as context switching functions provided by the sandbox mechanism. `unchecked`$(\overline{x})\{e\}$ is compiled to `sandbox_call`$(\overline{x}, e)$, where we call the sandbox to execute expression e with the arguments $\overline{x}$. `checked`$(\overline{x})\{e\}$ is compiled to `callback`$(\overline{x}, e)$, 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 dealt with 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 pointers [4]. 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 $(m, e)(\overline{e})$) is similar to the manipulation of pointer access operations in Figure 12.

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 (essentially, 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:

$$\mathcal{H} \gg \dot{\mathcal{H}}$$
$$\varphi \gg \dot{\varphi}$$

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 $\varphi \gg \dot{\varphi}$, $\mathcal{H} \gg \dot{\mathcal{H}}$

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

and $\Gamma; \Theta; \emptyset \vdash_m e \gg \dot{e} : \tau$ 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.[5] The simulation theorem is specified in a way that is similar to the one by Merigoux et al. [17].

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 $(\dot{\varphi}_1, \dot{\mathcal{H}}_1, \dot{e}_1) \dot{\rightarrow}^* (\dot{\varphi}, \dot{\mathcal{H}}, \dot{e})$.
- if $r_1 =$ `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 $\varphi_1 \gg \dot{\varphi}_1$, $\mathcal{H}_1 \gg \dot{\mathcal{H}}_1$.

Our random generator (discussed in the next section) never produces `unchecked` expressions (whose behavior could be undefined), so we can only test a the simulation theorem as it applies 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 arrangable. 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.

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

$$\begin{tikzcd}
\boxed{\varphi_0, \mathcal{H}_0, e_0} \arrow[rr, "\longrightarrow_c"] \arrow[d, leftrightarrow, "\gg"] & & \boxed{\varphi_1, \mathcal{H}_1, e_1} \arrow[d, leftrightarrow, "\gg"] \\
\boxed{\dot{\varphi}_0, \dot{\mathcal{H}}_0, \dot{e}_0} \arrow[rr, dashed, leftrightarrow, "\sim"] \arrow[dr, dashed, "\dot{\rightarrow}^*"] & & \boxed{\dot{\varphi}_1, \dot{\mathcal{H}}_1, \dot{e}_1} \arrow[dl, dashed, "\dot{\rightarrow}^*"'] \\
& \boxed{\dot{\varphi}, \dot{\mathcal{H}}, \dot{e}} &
\end{tikzcd}$$

Figure 14: Simulation between CORECHKCBOX and COREC

6. Related Work

Our work is most closely related to prior formalizations of C(-like) languages that aim to enforce 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 [1, 13], Ellison and Rosu [6], Kang et al. [11], and Memarian et al. [15, 16]. 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 be 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. [9] extends the formalization of Ellison and Rosu [6] 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 [20] and Softbound [18] 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 [8, 10] is a C dialect that aims to ensure memory safety; its pointer types are similar to CCured. Cyclone's formalization [8] focuses on the use of *regions* to ensure temporal safety; it does not formalize arrays or threats to spatial safety. Deputy [3, 28] is another safe-C dialect that aims to avoid fat pointers; it was an initial inspiration for Checked C's design [5], though it provides no specific modeling for null-terminated array pointers. Deputy's formalization [3] 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. [21]. They present the type system and semantics of a core model of Checked C, mechanized in Coq, and were the first to prove a blame theorem. CORECHKCBOX's Coq-based development (Section 3) substantially extends theirs to include conditionals, dynamically bounded array pointers with dependent types, null-terminated array pointers, dependently typed functions, and subtyping. They postulate that pointer metadata can be erased in a real implementation, but do not show it. Our CORECHKCBOX compiler, formalized and validated in PLT Redex via randomized testing, demonstrates that such metadata *can* be erased; we found that erasure was non-obvious once null-terminated pointers and bounds widening were considered.

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 testing 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 [12] or VeLLVM [27], 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] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009. ISBN 0262062755.

[8] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based Memory Management in Cyclone. In *PLDI*, 2002.

[9] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan 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.

[16] 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.

[17] Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. Catala: A Programming Language for the Law. *arXiv preprint arXiv:2103.03198*, 2021.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] CVE Trends. Cve trends. https://www.cvedetails.com/vulnerabilities-by-types.php, 2021. Accessed: 2020-10-11.

[26] 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.

[27] 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.

[28] 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 `EIfDef` 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 $e \equiv \texttt{let}\ x = (5 : \texttt{int})\ \texttt{in}\ x + x$. Expression e first steps to $\texttt{let}\ x = (5 : \texttt{int})\ \texttt{in}\ (5 : \texttt{int}) + x$, which in turns steps to $\texttt{let}\ x = (5 : \texttt{int})\ \texttt{in}\ (5 : \texttt{int}) + (5 : \texttt{int})$. Since the rhs of x is a value, the let binding in e effectively functions as a stack that maps from x to $5 : \texttt{int}$. 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., $\texttt{let}\ x = (5 : \texttt{int})\ \texttt{in}\ (10 : \texttt{int})$ steps to $10 : \texttt{int}$. 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 $\texttt{ptr}^{\texttt{c}}\ \omega$, 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 $[n, n + \mathit{size}(\omega))$, 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. [21].

3. Other Semantic Rules

Fig. 15 shows the remaining semantic rules for CORECHKCBOX. 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 $\texttt{ptr}^{\texttt{c}}\ \omega_a$ 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

$$\frac{}{(\varphi, \mathcal{H}, x) \longrightarrow (\varphi, \mathcal{H}, \varphi(x))}\ \text{S-VAR}$$

$$\frac{\mathcal{H}(n) = n_a : \tau_a \qquad 0 \in [n_l, n_h)}{(\varphi, \mathcal{H}, \texttt{*}\ n : \texttt{ptr}^c\ [(n_l, n_h)\ \tau]_{nt}) \longrightarrow (\varphi, \mathcal{H}, n_a : \tau)}\ \text{S-DEFARRAY}$$

$$\frac{0 \notin [n_l, n_h)}{(\varphi, \mathcal{H}, \texttt{*}\ n : \texttt{ptr}^c\ [(n_l, n_h)\ \tau]_\kappa) \longrightarrow (\varphi, \mathcal{H}, \texttt{bounds})}\ \text{S-DEFARRAYBOUND}$$

$$\frac{0 \notin [n_l, n_h]}{(\varphi, \mathcal{H}, \texttt{*}\ n : \texttt{ptr}^c\ [(n_l, n_h)\ \tau]_{nt}) \longrightarrow (\varphi, \mathcal{H}, \texttt{bounds})}\ \text{S-DEFNTARRAYBOUND}$$

$$\frac{\mathcal{H}(n) = n_a : \tau_a}{(\varphi, \mathcal{H}, \texttt{*}\ n : \texttt{ptr}^c\ \tau \texttt{=} n_1 : \tau_1) \longrightarrow (\varphi, \mathcal{H}[n \mapsto n_1 : \tau], n_1 : \tau)}\ \text{S-ASSIGN}$$

$$\frac{}{(\varphi, \mathcal{H}, \texttt{*}\ 0 : \texttt{ptr}^c\ \omega \texttt{=} n_1 : \tau_1) \longrightarrow (\varphi, \mathcal{H}, \texttt{null})}\ \text{S-ASSIGNNULL}$$

$$\frac{0 \notin [n_l, n_h)}{(\varphi, \mathcal{H}, \texttt{*}\ n : \texttt{ptr}^c\ [(n_l, n_h)\ \tau]_\kappa \texttt{=} n_1 : \tau_1) \longrightarrow (\varphi, \mathcal{H}, \texttt{bounds})}\ \text{S-ASSIGNARRBOUND}$$

$$\frac{\varphi(\omega) = \omega_a \qquad \texttt{alloc}(\mathcal{H}, \omega_a) = (n, \mathcal{H}')}{(\varphi, \mathcal{H}, \texttt{malloc}(\omega,)) \longrightarrow (\varphi, \mathcal{H}', n : \texttt{ptr}^c\ \omega_a)}\ \text{S-MALLOC}$$

$$\frac{\varphi(\omega) = [(n_l, n_h)\ \tau]_\kappa \qquad (n_l \neq 0 \vee n_h \leq 0)}{(\varphi, \mathcal{H}, \texttt{malloc}(\omega,)) \longrightarrow (\varphi, \mathcal{H}', \texttt{bounds})}\ \text{S-MALLOCBOUND}$$

$$\frac{n \neq 0}{(\varphi, \mathcal{H}, \texttt{if}\ (n : \tau)\ e_1\ \texttt{else}\ e_2) \longrightarrow (\varphi, \mathcal{H}, e_1)}\ \text{S-IFT}$$

$$\frac{}{(\varphi, \mathcal{H}, \texttt{if}\ (0 : \tau)\ e_1\ \texttt{else}\ e_2) \longrightarrow (\varphi, \mathcal{H}, e_2)}\ \text{S-IFF}$$

$$\frac{}{(\varphi, \mathcal{H}, \texttt{unchecked}(n : \tau)\{\longrightarrow\}(\varphi, \mathcal{H}, n : \tau)}\ \text{S-UNCHECKED}$$

$$\frac{0 \in [n_l, n_h] \qquad n_a \leq n_h \qquad \mathcal{H}(n + n_a) = 0 \qquad (\forall i. n \leq i < n + n_a \Rightarrow (\exists n_i\ t_i . \mathcal{H}(n + i) = n_i : \tau_i \wedge n_i \neq 0))}{(\varphi, \mathcal{H}, \texttt{strlen}(n : \texttt{ptr}^m\ [(n_l, n_h)\ \tau])) \longrightarrow (\varphi, \mathcal{H}, n_a : \texttt{int})}\ \text{S-STR}$$

$$\frac{0 \notin [n_l, n_h]}{(\varphi, \mathcal{H}, \texttt{strlen}(n : \texttt{ptr}^c\ [(n_l, n_h)\ \tau])) \longrightarrow (\varphi, \mathcal{H}, \texttt{bounds})}\ \text{S-STRBOUNDS}$$

$$\frac{}{(\varphi, \mathcal{H}, \texttt{strlen}(0 : \texttt{ptr}^c\ [(n_l, n_h)\ \tau])) \longrightarrow (\varphi, \mathcal{H}, \texttt{null})}\ \text{S-STRNULL}$$

$$\frac{n = n_1 + n_2}{(\varphi, \mathcal{H}, n_1 : \texttt{int} \texttt{+} n_2 : \texttt{int}) \longrightarrow (\varphi, \mathcal{H}, n)}\ \text{S-ADD}$$

$$\frac{n = n_1 + n_2 \qquad n_l' = n_l - n_2 \qquad n_h' = n_h - n_2}{(\varphi, \mathcal{H}, n_1 : \texttt{ptr}^m\ [(n_l, n_h)\ \tau]_\kappa \texttt{+} n_2 : \texttt{int}) \longrightarrow (\varphi, \mathcal{H}, n : \texttt{ptr}^m\ [(n_l', n_h')\ \tau]_\kappa)}\ \text{S-ADDARR}$$

$$\frac{}{n(\varphi, \mathcal{H}, 0 : \texttt{ptr}^c\ [(n_l, n_h)\ \tau]_\kappa \texttt{+} n_2 : \texttt{int}) \longrightarrow (\varphi, \mathcal{H}, \texttt{null})}\ \text{S-ADDARRNULL}$$

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

relation: $\varphi|_\rho \subseteq \varphi$. $\sqsubseteq$ 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 $\texttt{ptr}^c\ [(0, x)\ \texttt{int}]_{nt}$ 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 $\texttt{ptr}^c\ [(0, x)\ \texttt{int}]_{nt} \sqsubseteq \texttt{ptr}^c\ [(0, 0)\ \texttt{int}]_{nt}$ 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 $\leq$ comparasion 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.

$$\frac{\Gamma;\Theta\vdash_m e : \texttt{ptr}^{m'}\ \tau \qquad m \le m'}{\Gamma;\Theta\vdash_m \texttt{*}\ e : \tau}\ \text{T-DEF}$$

$$\Gamma;\Theta\vdash_m \texttt{malloc}(\omega,:)\texttt{ptr}^{c}\ \omega\ \text{T-MAC}$$

$$\frac{\Gamma;\Theta\vdash_m e_1 : \texttt{int} \qquad \Gamma;\Theta\vdash_m e_2 : \texttt{int}}{\Gamma;\Theta\vdash_m (e_1 \texttt{+} e_2) : \texttt{int}}\ \text{T-ADD}$$

$$\frac{\Gamma;\Theta\vdash_m e_1 : \texttt{ptr}^{m'}\ [\beta\ \tau]_\kappa \qquad \Gamma;\Theta\vdash_m e_2 : \texttt{int} \qquad m \le m'}{\Gamma;\Theta\vdash_m \texttt{*}\ (e_1 \texttt{+} e_2) : \tau}\ \text{T-IND}$$

$$\frac{\Gamma;\Theta\vdash_m e_1 : \texttt{ptr}^{m'}\ \tau \qquad \Gamma;\Theta\vdash_m e_2 : \tau' \qquad \tau' \sqsubseteq \tau \qquad m \le m'}{\Gamma;\Theta\vdash_m \texttt{*}\ e_1 \texttt{=} e_2 : \tau}\ \text{T-ASSIGN}$$

$$\frac{\Gamma;\Theta\vdash_m e_1 : \texttt{ptr}^{m'}\ [\beta\ \tau]_\kappa \qquad \Gamma;\Theta\vdash_m e_2 : \texttt{int} \qquad \Gamma;\Theta\vdash_m e_3 : \tau' \qquad \tau' \sqsubseteq \tau \qquad m \le m'}{\Gamma;\sigma\vdash_m \texttt{*}\ (e_1 \texttt{+} e_2) \texttt{=} e_3 : \tau}\ \text{T-INDASSIGN}$$

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

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 $m \le m'$ 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 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 $\texttt{ptr}^m$ `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.

Struct Syntax:

Type	$\texttt{struct}\ T$
Structdefs	$D \in T \rightharpoonup fs$
Fields	$fs ::= \tau\ \texttt{f} \mid \tau\ \texttt{f}; fs$

Struct Subtype:

$$D(T) = fs \wedge fs(0) = \texttt{nat} \Rightarrow \texttt{ptr}^m\ \texttt{struct}\ T \sqsubseteq \texttt{ptr}^m\ \texttt{nat}$$

$$D(T) = fs \wedge fs(0) = \texttt{nat} \wedge 0 \le b_l \wedge b_h \le 1 \Rightarrow \texttt{ptr}^m\ \texttt{struct}\ T \sqsubseteq \texttt{ptr}^m\ [(b_l, b_h)\ \texttt{nat}]$$

Struct Type Rule:

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

Struct Semantics:

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

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

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

Figure 17: CORECHKCBOX Struct Definitions

$$\Gamma \vdash n \qquad \frac{x : \texttt{int} \in \Gamma}{\Gamma \vdash x + n} \qquad \frac{\Gamma \vdash b_l \qquad \Gamma \vdash b_h}{\Gamma \vdash (b_l, b_h)} \qquad \Gamma \vdash \texttt{int}$$

$$\frac{\Gamma \vdash \beta \qquad \Gamma \vdash \tau}{\Gamma \vdash \texttt{ptr}^m\ [\beta\ \tau]_\kappa} \qquad \frac{\Gamma \vdash \tau}{\Gamma \vdash \texttt{ptr}^m\ \tau} \qquad \frac{T \in D}{\Gamma \vdash \texttt{ptr}^m\ \texttt{struct}\ T}$$

Figure 18: Well-formedness for Types and Bounds

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 oc-

$$\frac{\Gamma \vdash \overline{x} : \overline{\tau} \qquad \Gamma[\overline{x} \mapsto \overline{\tau}] \vdash \tau \qquad \Gamma[\overline{x} \mapsto \overline{\tau}]; \Theta \vdash_{\mathsf{c}} e : \tau}{\Gamma \vdash \tau\ (\overline{x} : \overline{\tau})\ e} \qquad \Gamma \vdash \cdot$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma[x \mapsto \tau] \vdash \overline{x} : \overline{\tau}}{\Gamma \vdash x : \tau, \overline{x} : \overline{\tau}}$$

Figure 19: Well-formedness for functions

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau\ \mathtt{f}} \qquad \frac{\Gamma \vdash \tau \qquad \Gamma \vdash fs}{\Gamma \vdash \tau\ \mathtt{f}; fs}$$

Figure 20: Well-formedness for structs

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

Figure 21: Compilation Rules for Functions

casionally 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 need to halt with a bounds error or widen a null-terminated string.

$$\begin{array}{llll}
\text{Atoms} & \dot{a} & ::= & n \mid x \\
\text{C-Expressions} & \dot{c} & ::= & \dot{a} \mid \mathtt{strlen}(\dot{a}) \mid \mathtt{malloc}(\dot{a}, |) f(\overline{\dot{a}}) \\
 & & \mid & \mid \dot{a} \circ \dot{a} \mid \mathtt{*}\ \dot{a} \\
 & & \mid & \mathtt{*}\ \dot{a} = \dot{a} \mid x = \dot{a} \mid \mathtt{if}\ (\dot{a})\ \dot{e}\ \mathtt{else}\ \dot{e} \\
 & & \mid & \mathtt{bounds} \mid \mathtt{null} \\
\text{Expressions} & \dot{e} & ::= & \dot{c} \mid \mathtt{let}\ x = \dot{c}\ \mathtt{in}\ \dot{e} \\
\text{Binops} & \circ & ::= & + \mid - \mid \leq \\
\text{Closure} & \dot{C} & ::= & \Box \mid \mathtt{let}\ x = \dot{a}\ \mathtt{in}\ \dot{C} \\
 & & \mid & \mathtt{if}\ (\dot{a})\ \dot{e}\ \mathtt{else}\ \dot{C} \mid \mathtt{if}\ (\dot{a})\ \dot{C}\ \mathtt{else}\ \dot{e} \\
\text{Bounds Map} & \rho & \in & \mathtt{Var} \rightharpoonup \mathtt{Var} \times \mathtt{Var}
\end{array}$$

Figure 22: COREC Syntax

$$\begin{array}{lll}
\dot{\mu} & ::= & n \mid \bot \\
\dot{c} & ::= & \ldots \mid \mathtt{ret}(x, \dot{\mu}, \dot{e}) \\
\dot{H} & \in & \mathbb{Z} \rightharpoonup \mathbb{Z} \\
\dot{r} & ::= & \dot{e} \mid \mathtt{null} \mid \mathtt{bounds} \\
\dot{E} & ::= & \Box \mid \mathtt{let}\ x = \dot{E}\ \mathtt{in}\ \dot{e} \mid \mathtt{ret}(x, i, \dot{E}) \\
 & \mid & \mathtt{if}\ (\dot{E})\ \dot{e}\ \mathtt{else}\ \dot{e} \mid \mathtt{strlen}(\dot{E}) \\
 & \mid & \mathtt{malloc}(\dot{E}, |) f(\overline{\dot{E}}) \mid \dot{E} \circ \dot{a} \mid n \circ \dot{E} \\
 & \mid & \mathtt{*}\ \dot{E} \mid \mathtt{*}\ \dot{E} = \dot{a} \mid \mathtt{*}\ n = \dot{E} \mid x = \dot{E} \\
\overline{\dot{E}} & ::= & \dot{E} \mid n, \overline{\dot{E}} \mid \overline{\dot{E}}, \dot{a}
\end{array}$$

Figure 23: COREC Semantic Defs

COREC does not include any annotations. We remove structs from COREC because we can always statically convert expressions of the form $\&n : \tau \to f$ into $n + n_f$, where n_f is the statically determined offset of f within the struct. 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 $(1 : \mathtt{int}) + (2 : \mathtt{int})$, we would first create a fresh variable x, and then produce two outputs:

$$\dot{C} = \mathtt{let}\ x = 1 + 2\ \mathtt{in}\ \Box$$

$$\dot{a} = x$$

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 $\mathtt{let}\ x = \dot{c}\ \mathtt{in}\ \Box$, where $\dot{c}$ contains the check whose evaluation must not 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.

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 $\Box$ 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 retrived 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.

$$\frac{x_l, x_h = \texttt{fresh} \qquad \rho' = \rho[x \mapsto (x_l, x_h)] \qquad \dot{C} = \texttt{let}\ x_l \,\texttt{=}\, b_l\ \texttt{in let}\ x_h \,\texttt{=}\, b_h\ \texttt{in}\ \Box}{\dot{C}, \rho' = \ \vdash_{extend} \rho, x, \texttt{ptr}^c\ [(b_l, b_h)\ \tau]_{nt}}$$

$$\frac{x_l, x_h = \rho(x) \qquad x_w = \texttt{fresh} \qquad \dot{C} = \texttt{let}\ x_w \,\texttt{=}\, \texttt{if}\ (x_h)\ 0\ \texttt{else}\ x_h \,\texttt{=}\, 1\ \texttt{in}\ \Box}{\dot{C} = \ \vdash_{widenderef} \rho, x, \texttt{ptr}^c\ [(b_l, b_h)\ \tau]_{nt}} \qquad \frac{e \notin dom(\rho)}{\Box = \ \vdash_{widenstr} \rho, e, \dot{a}, \texttt{ptr}^m\ [\beta\ \tau]_{nt}}$$

$$\frac{x_l, x_h = \rho(e) \qquad x_a = \texttt{fresh} \qquad \dot{C} = \texttt{let}\ x_a \,\texttt{=}\, \texttt{if}\ (\dot{a} \le x_h)\ 0\ \texttt{else}\ x_h \,\texttt{=}\, \dot{a}\ \texttt{in}\ \Box}{\dot{C} = \ \vdash_{widenstr} \rho, e, \dot{a}, \texttt{ptr}^c\ [\beta\ \tau]_{nt}}$$

Figure 24: Metafunctions for Widening

$$\frac{x = \texttt{fresh} \qquad \dot{C} = \texttt{let}\ x \,\texttt{=}\, \texttt{if}\ (\dot{a})\ 0\ \texttt{else null in}\ \Box}{\dot{C} = \ \vdash_{null} \dot{a}, c} \qquad \Box = \ \vdash_{null} \dot{a}, u$$

$$\Box = \ \vdash_{boundsR} \rho, e, \texttt{ptr}^u\ [\beta\ \tau]_\kappa, \dot{a}$$

$$\frac{\begin{array}{c} x_l, x_h = \rho(e) \\ x_{cl}, x_{ch} = \texttt{fresh} \qquad \dot{C}_{cl} = \texttt{let}\ x_{cl} \,\texttt{=}\, \texttt{if}\ (x_l \le \dot{a})\ 0\ \texttt{else bounds in}\ \Box \qquad \dot{C}_{ch} = \texttt{let}\ x_{ch} \,\texttt{=}\, \texttt{if}\ (\dot{a} \le x_h)\ 0\ \texttt{else bounds in}\ \Box \end{array}}{\dot{C}_{cl;ch} = \ \vdash_{boundsR} \rho, e, \texttt{ptr}^c\ [\beta\ \tau]_\kappa, \dot{a}}$$

$$\frac{\begin{array}{c} e \notin dom(\rho) \qquad x_l, x_h, x_{cl}, x_{ch} = \texttt{fresh} \qquad \dot{C}_l = \texttt{let}\ x_l \,\texttt{=}\, b_l\ \texttt{in}\ \Box \qquad \dot{C}_h = \texttt{let}\ x_h \,\texttt{=}\, b_h\ \texttt{in}\ \Box \\ \dot{C}_{cl} = \texttt{let}\ x_{cl} \,\texttt{=}\, \texttt{if}\ (x_l \le \dot{a})\ 0\ \texttt{else bounds in}\ \Box \qquad \dot{C}_{ch} = \texttt{let}\ x_{ch} \,\texttt{=}\, \texttt{if}\ (\dot{a} \le x_h)\ 0\ \texttt{else bounds in}\ \Box \end{array}}{\dot{C}_{l;h;cl;ch} = \ \vdash_{boundsR} \rho, e, \texttt{ptr}^c\ [(b_l, b_h)\ \tau]_{nt}, \dot{a}}$$

$$\frac{\begin{array}{c} e \notin dom(\rho) \qquad x_l, x_h, x_{cl}, x_{ch} = \texttt{fresh} \qquad \dot{C}_l = \texttt{let}\ x_l \,\texttt{=}\, b_l\ \texttt{in}\ \Box \qquad \dot{C}_h = \texttt{let}\ x_h \,\texttt{=}\, b_h\ \texttt{in}\ \Box \\ \dot{C}_{cl} = \texttt{let}\ x_{cl} \,\texttt{=}\, \texttt{if}\ (x_l \le \dot{a})\ 0\ \texttt{else bounds in}\ \Box \qquad \dot{C}_{ch} = \texttt{let}\ x_{ch} \,\texttt{=}\, \texttt{if}\ (x_h \le \dot{a})\ \texttt{bounds else}\ 0\ \texttt{in}\ \Box \end{array}}{\dot{C}_{l;h;cl;ch} = \ \vdash_{boundsR} \rho, e, \texttt{ptr}^c\ [(b_l, b_h)\ \tau], \dot{a}}$$

$$\Box = \ \vdash_{boundsW} \rho, e, \texttt{ptr}^u\ [\beta\ \tau]_\kappa, \dot{a}$$

$$\frac{\begin{array}{c} x_l, x_h = \rho(e) \\ x_{cl}, x_{ch} = \texttt{fresh} \qquad \dot{C}_{cl} = \texttt{let}\ x_{cl} \,\texttt{=}\, \texttt{if}\ (x_l \le \dot{a})\ 0\ \texttt{else bounds in}\ \Box \qquad \dot{C}_{ch} = \texttt{let}\ x_{ch} \,\texttt{=}\, \texttt{if}\ (\dot{a} \le x_h)\ 0\ \texttt{else bounds in}\ \Box \end{array}}{\dot{C}_{cl;ch} = \ \vdash_{boundsW} \rho, e, \texttt{ptr}^c\ [\beta\ \tau]_\kappa, \dot{a}}$$

$$\frac{\begin{array}{c} e \notin dom(\rho) \qquad x_l, x_h, x_{cl}, x_{ch} = \texttt{fresh} \qquad \dot{C}_l = \texttt{let}\ x_l \,\texttt{=}\, b_l\ \texttt{in}\ \Box \qquad \dot{C}_h = \texttt{let}\ x_h \,\texttt{=}\, b_h\ \texttt{in}\ \Box \\ \dot{C}_{cl} = \texttt{let}\ x_{cl} \,\texttt{=}\, \texttt{if}\ (x_l \le \dot{a})\ 0\ \texttt{else bounds in}\ \Box \qquad \dot{C}_{ch} = \texttt{let}\ x_{ch} \,\texttt{=}\, \texttt{if}\ (x_h \le \dot{a})\ \texttt{bounds else}\ 0\ \texttt{in}\ \Box \end{array}}{\dot{C}_{l;h;cl;ch} = \ \vdash_{boundsW} \rho, e, \texttt{ptr}^c\ [(b_l, b_h)\ \tau]_\kappa, \dot{a}}$$

$$\frac{\begin{array}{c} e \notin dom(\rho) \qquad x_l, x_l', x_h, x_h' = \texttt{fresh} \qquad \dot{C}_1 = \texttt{let}\ x_l \,\texttt{=}\, b_l\ \texttt{in let}\ x_h \,\texttt{=}\, b_h\ \texttt{in}\ \Box \\ \dot{C}_2 = \texttt{let}\ x_l' \,\texttt{=}\, b_l'\ \texttt{in let}\ x_h' \,\texttt{=}\, b_h'\ \texttt{in}\ \Box \qquad \dot{C}_3 = \texttt{if}\ (x_l' \le x_l)\ \Box\ \texttt{else bounds} \qquad \dot{C}_4 = \texttt{if}\ (x_h \le x_h')\ \Box\ \texttt{else bounds} \end{array}}{\dot{C}_{1;2;3;4} = \ \vdash_{boundsD} \rho, e, \texttt{ptr}^m\ [(b_l, b_h)\ \tau]_\kappa, \texttt{ptr}^m\ [(b_l', b_h')\ \tau]_\kappa}$$

$$\frac{\begin{array}{c} x_l', x_h' = \rho(e) \qquad x_l, x_h = \texttt{fresh} \\ \dot{C}_1 = \texttt{let}\ x_l \,\texttt{=}\, b_l\ \texttt{in let}\ x_h \,\texttt{=}\, b_h\ \texttt{in}\ \Box \qquad \dot{C}_2 = \texttt{if}\ (x_l' \le x_l)\ \Box\ \texttt{else bounds} \qquad \dot{C}_3 = \texttt{if}\ (x_h \le x_h')\ \Box\ \texttt{else bounds} \end{array}}{\dot{C}_{1;2;3} = \ \vdash_{boundsD} \rho, e, \texttt{ptr}^m\ [(b_l, b_h)\ \tau]_\kappa, \texttt{ptr}^m\ [(b_l', b_h')\ \tau]_\kappa}$$

Figure 25: Metafunctions for Dynamic Checks

C-CONST

$$\Gamma;\rho \vdash n:\tau \gg \square, n:\tau$$

C-VAR

$$\frac{x:\tau\in\Gamma}{\Gamma;\rho\vdash x \gg \square, x:\tau}$$

C-CAST

$$\frac{\Gamma;\rho\vdash e\gg \dot{C},\dot{a}:\tau'}{\Gamma;\rho\vdash(\tau)e\gg\dot{C},\dot{a}:\tau}$$

C-DYNCAST

$$\frac{\Gamma;\rho\vdash e\gg\dot{C}_1,\dot{a}:\texttt{ptr}^m\ [\beta'\ \tau]_\kappa \qquad \dot{C}_b = \ \vdash_{boundsD}\rho,e,\texttt{ptr}^m\ [\beta\ \tau]_\kappa,\texttt{ptr}^m\ [\beta'\ \tau]_\kappa}{\Gamma;\rho\vdash\langle\texttt{ptr}^m\ [\beta\ \tau]_\kappa\rangle e\gg\dot{C}_{1;b},\dot{a}:\texttt{ptr}^m\ [\beta\ \tau]_\kappa}$$

C-STR

$$\frac{\begin{array}{c}\Gamma;\rho\vdash e\gg\dot{C}_1,\dot{a}_1:\texttt{ptr}^m\ [\beta\ \tau_a]_{nt} \qquad \dot{C}_n = \ \vdash_{null}\dot{a}_1,m \qquad \dot{C}_b = \ \vdash_{boundsR}\rho,\dot{a}_1,\texttt{ptr}^m\ [\beta\ \tau_a]_{nt},0 \\ x_2=\texttt{fresh} \qquad \dot{C}_2 = \texttt{let}\ x_2\,\texttt{=}\,\texttt{strlen}(\dot{a}_1)\ \texttt{in}\ \square \qquad \dot{C}_w = \ \vdash_{widenstr}\rho,e,\dot{a}_1,\texttt{ptr}^m\ [\beta\ \tau_a]_{nt}\end{array}}{\Gamma;\rho\vdash\texttt{strlen}(e)\gg\dot{C}_{1;n;b;2;w},x_2:\texttt{int}}$$

C-LETSTR

$$\frac{\begin{array}{c}\Gamma(y)=\texttt{ptr}^c\ [(b_l,b_h)\ \tau_a]_{nt} \qquad x\notin FV(\tau) \\ \Gamma;\rho\vdash\texttt{strlen}(y)\gg\dot{C}_1,\dot{a}_1:\texttt{int} \qquad \dot{C}_2=\texttt{let}\ x\,\texttt{=}\,\dot{a}_1\ \texttt{in}\ \square \qquad \Gamma[x\mapsto\texttt{int},y\mapsto[\texttt{ptr}^c\ [(b_l,x)\ \tau_a]_{nt}]];\rho\vdash e_3\gg\dot{C}_3,\dot{a}_3:\tau\end{array}}{\Gamma;\rho\vdash\texttt{let}\ x\,\texttt{=}\,\texttt{strlen}(y)\ \texttt{in}\ e\gg\dot{C}_{1;2;3},\dot{a}_3:\tau}$$

C-IF

$$\frac{\begin{array}{c}\Gamma;\rho\vdash e\gg\dot{C}_1,\dot{a}_1:\tau \\ \Gamma;\rho\vdash e_1\gg\dot{C}_2,\dot{a}_2:\tau_2 \qquad \Gamma;\rho\vdash e_3\gg\dot{C}_3,\dot{a}_3:\tau_3 \qquad x_4=\texttt{fresh} \qquad \dot{C}_4=\texttt{let}\ x_4\,\texttt{=}\,\texttt{if}\ (\dot{a}_1)\ \dot{C}_2[\dot{a}_2]\ \texttt{else}\ \dot{C}_3[\dot{a}_3]\ \texttt{in}\ \square\end{array}}{\Gamma;\rho\vdash\texttt{if}\ (e_1)\ e_2\ \texttt{else}\ e_3\gg\dot{C}_{1;4},x_4:\tau_2\sqcup\tau_3}$$

C-IFNT

$$\frac{\begin{array}{c}\Gamma;\rho\vdash x:\texttt{ptr}^c\ [(b_l,b_h)\ \tau]_{nt} \qquad b_h=0\Rightarrow\Gamma'=\Gamma[x\mapsto\texttt{ptr}^c\ [(b_l,1)\ \tau]_{nt}] \\ b_h\neq 0\Rightarrow\Gamma'=\Gamma \qquad \Gamma;\rho\vdash\texttt{*}\ x\gg\dot{C}_1,\dot{a}_1:\tau_1 \qquad \Gamma';\rho\vdash e_2\gg\dot{C}_2,\dot{a}_2:\tau_2 \qquad \Gamma;\rho\vdash e_3\gg\dot{C}_3,\dot{a}_3:\tau_3 \\ \dot{C}_w = \ \vdash_{widenderef}\rho,x,\texttt{ptr}^c\ [(b_l,b_h)\ \tau]_{nt} \qquad x_4=\texttt{fresh} \qquad \dot{C}_4=\texttt{let}\ x_4\,\texttt{=}\,\texttt{if}\ (\dot{a}_1)\ \dot{C}_{2;w}[\dot{a}_2]\ \texttt{else}\ \dot{C}_3[\dot{a}_3]\ \texttt{in}\ \square\end{array}}{\Gamma;\rho\vdash\texttt{if}\ (\texttt{*}\ x)\ e_1\ \texttt{else}\ e_2\gg\dot{C}_{1;4},x_4:\tau_1\sqcup\tau_2}$$

C-LET

$$\frac{\begin{array}{c}(x\in FV(\tau')\Rightarrow e_1\in \mathit{Bound}) \\ \Gamma;\rho\vdash e_1\gg\dot{C}_1,\dot{a}_1:\tau_1 \qquad \dot{C}_2,\rho' = \ \vdash_{extend}\rho,x,\tau_1 \qquad \dot{C}_3=\texttt{let}\ x\,\texttt{=}\,\dot{a}_1\ \texttt{in}\ \square \qquad \Gamma[x\mapsto\tau];\rho'\vdash e_4\gg\dot{C}_4,\dot{a}_4:\tau_4\end{array}}{\Gamma;\rho'\vdash\texttt{let}\ x\,\texttt{=}\,e_1\ \texttt{in}\ e_4\gg\dot{C}_{1;2;3;4},\dot{a}_4:\tau_4[\tau_1=\texttt{int}\Rightarrow x\mapsto e_1]}$$

C-RET

$$\frac{\Gamma(x)\neq\bot \qquad \Gamma;\rho\vdash e\gg\dot{C}_1,\dot{a}_1:\tau \qquad x_2=\texttt{fresh} \qquad \mu\gg\dot{\mu} \qquad \dot{C}_2=\texttt{let}\ x_2\,\texttt{=}\,\texttt{ret}(x,\dot{\mu},\dot{C}_1[\dot{a}_1])\ \texttt{in}\ \square}{\Gamma;\rho\vdash\texttt{ret}(x,\mu,e)\gg\dot{C}_2,x_2:\tau}$$

C-FUN

$$\frac{\Xi(f)=\tau\ (\overline{x}:\overline{\tau})\ e \qquad (\forall e_i\in\overline{e}\ \ \tau_i\in\overline{\tau}\ .\ \Gamma;\rho\vdash e_i\gg\dot{C}_i,\dot{a}_i: \quad \tau_i'\wedge\tau_i'\sqsubseteq\tau_i[\overline{e}/\overline{x}]) \qquad x_f=\texttt{fresh} \qquad \dot{C}_f=\texttt{let}\ x_f\,\texttt{=}\,f(\overline{a})\ \texttt{in}\ \square}{\Gamma;\rho\vdash f(\overline{e})\gg\overline{\dot{C}}[\dot{C}_f],x_f:\tau[\overline{e}/\overline{x}]}$$

C-DEF

$$\frac{\Gamma;\rho\vdash e_1\gg\dot{C}_1,\dot{a}_1:\texttt{ptr}^m\ \tau \qquad \dot{C}_n = \ \vdash_{null}\dot{a}_1,m \qquad x_2=\texttt{fresh} \qquad \dot{C}_2=\texttt{let}\ x_2\,\texttt{=}\,\texttt{*}\ \dot{a}_1\ \texttt{in}\ \square}{\Gamma;\rho\vdash\texttt{*}\ e_1\gg\dot{C}_{1;n;2},x_2:\tau}$$

C-DEFARR

$$\frac{\begin{array}{c}\Gamma;\rho\vdash e_1\gg\dot{C}_1,\dot{a}_1:\texttt{ptr}^m\ [(b_l,b_h)\ \tau]_\kappa \\ \dot{C}_n = \ \vdash_{null}\dot{a}_1,m \qquad \dot{C}_b = \ \vdash_{boundsR}\rho,e_1,\texttt{ptr}^m\ [(b_l,b_h)\ \tau]_\kappa,0 \qquad x_2=\texttt{fresh} \qquad \dot{C}_2=\texttt{let}\ x_2\,\texttt{=}\,\texttt{*}\ \dot{a}_1\ \texttt{in}\ \square\end{array}}{\Gamma;\rho\vdash\texttt{*}\ e_1\gg\dot{C}_{1;n;b;2},x_2:\tau}$$

C-MAC

$$\frac{\dot{C}_1,\dot{a}_1=\texttt{sizeof}(\omega) \qquad x_2=\texttt{fresh} \qquad \dot{C}_2=\texttt{let}\ x_2\,\texttt{=}\,\texttt{malloc}(\dot{a}_1,\ )\texttt{in}\ \square}{\Gamma;\rho\vdash\texttt{malloc}(\omega,\gg)\dot{C}_{1;2},x_2:\texttt{ptr}^c\ \omega}$$

Figure 26: Compilation

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

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

C-ASSIGN
$$\frac{\begin{array}{c}\Gamma;\rho\vdash e_1 \gg \dot{C}_1,\dot{a}_1:\texttt{ptr}^{m'}\ \tau\\ \dot{C}_n=\ \vdash_{null}\dot{a}_1,m\qquad \Gamma;\rho\vdash e_2 \gg \dot{C}_2,\dot{a}_2:\tau'\qquad \tau'\sqsubseteq\tau\qquad x_3=\texttt{fresh}\qquad \dot{C}_3=\texttt{let}\ x_3\,\texttt{=}\,\texttt{*}\ \dot{a}_1\,\texttt{=}\,\dot{a}_2\ \texttt{in}\ \Box\end{array}}{\Gamma;\rho\vdash \texttt{*}\ e_1\,\texttt{=}\,e_2\gg \dot{C}_{1;2;n;3},x_3:\tau}$$

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

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

C-STRUCT
$$\frac{\begin{array}{c}\Gamma;\rho\vdash e_1 \gg \dot{C}_1,\dot{a}_1:\texttt{ptr}^m\ \texttt{struct}\ T\\ D(T)=\tau_0\ f_0\ldots;\tau_j\ f;\ldots\qquad \dot{C}_n=\ \vdash_{null}\dot{a}_1,m\qquad x_2=\texttt{fresh}\qquad \dot{C}_2=\texttt{let}\ x_2\,\texttt{=}\,\dot{a}_1\,\texttt{+}\,\dot{j}\ \texttt{in}\ \Box\end{array}}{\Gamma;\rho\vdash \&e_1\to f\gg \dot{C}_2,x_2:\texttt{ptr}^m\ \tau_f}$$

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

Figure 27: Compilation (Continued)