**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.

# Introduction

In C applications spatial memory corruption is was a main issue with various vulnerabilities and exploitation possibilities. Various steps were taken to exclude this risk with the help of different languages which were extremely challenging. The deployment cost also increased eventually which cannot suit a low powered devices. We used the conversion of C to Checked C but still there were some limitations such as safety issues among the unchecked code and much more.

Spatial memory corruption vulnerabilities remain a prominent issue for C applications [3, 42], despite mitigation efforts [39]. Converting present apps to safe languages like Java is challenging and not (yet) practical for performance-critical settings like Embedded OS. High-performance secure system languages like Rust [28] and Go [43] exist, but they're too different to be practical. Existing automatic conversion tools [5, 11, 17] can only take modest steps (to unsafe, non-idiomatic Rust). CCured [33], Softbound [29], Low Fat pointers [8], and Address Sanitizer (ASAN) [37] analyze and build C programs to add run-time safety checks. Run-time costs are too high for deployment (60-200%), especially in low-powered IoT devices using C-based system software (e.g., FreeRTOS).

**Checked C.** Checked C prevents spatial memory issues (temporal safety is underway [46]) with no overhead, Tarditi et al. developed *Checked C*. Checked C adds checked pointer types that the compiler restricts to safe uses (temporal safety is under- way [46]). Such pointers have one of three possible types, ptr<T >, array\_ptr<T >, or nt\_array\_ptr<T > (ptr, arr, and ntarr for short), representing a pointer to a single element, array of elements, or null-terminated array of elements of type T , respectively. Latter two have an associated bounds annotation; e.g., a declaration array\_ptr<int>p : count(n) says that p is a pointer to an int array whose size is n. To prevent spatial safety breaches, e compiler adds dynamic checks before verified pointer accesses. LLVM removes run-time redundancies, enhancing performance. Duan et al. [7] discovered no run-time overhead when running Checked C-converted FreeBSD kernel components. Checked C maintains backward compatibility by encoding checked pointers as system-level memory words. Some pointers with Checked annotations can be converted incrementally or partially. Partially annotated programs are safe using Checked pointers (i.e., checked or safe regions). Figure 1 compares Checked C to existing methods. Converting to Checked C increases the safety of C system software.

**Converting C to Checked C.** There are certain limitations to Checked C's safety promises. For instance, as illustrated below, Checked C programs are not allowed to utilize address-taken variables in a boundaries expression because pointers may have modified the bounds relations, making it possible for them to break.

...

array\_ptr<int>p: count (n)=NULL;

X..,&n,.

Converting existing C programs to Checked C may need rewriting, e.g., removing the &n expression from the above program [7]. Machiry et al. recently developed 3C [23], which adds pointer annotations to convert a program to Checked C. According to 3C, total automated conversion isn't possible and some code must be converted manually. Backward compatibility allows a partially annotated program to use only Checked pointers in memory-safe parts (i.e., checked or safe regions).

**No safety against unchecked code.** But, the unconverted code regions (or unsafe regions) can affect pointers in safe regions and violate certain assumptions leading to vulnerabilities, as demonstrated by cross-language attacks [26]. Al- though the blameless proof exists [35], it does not state that spatial safety violations cannot happen in Checked regions but rather states that Checked regions *cannot be blamed for any spatial safety violations*.

Consider the following example:

1 // Checked code

2 intfunc(array\_ptr<char>p: count(5)) {

3 Š..p[4]..

4 }

5 // unchecked code

6 ...

7 str="he";

8 ...

9 Wassume\_bounds\_cast<char>(str,5);

10 ...

11 charptr[16];

12 ...

13 len<-derived from user input

14 ...

15 W memcpy(ptr, buff, len);// buffer overflow

Unchecked code invoked the checked function func with a buffer of 2 items. This causes a spatial safety violation () in the Checked region, but the unchecked region is to blame (W). Since checked and unchecked regions operate in the same address space, spatial memory corruptions in unchecked regions (Line 15) can knock down the entire program. Unchecked code must not violate checked code's safety requirements, hence we need an isolation technique.

[1, 4, 21, 22, 40] Study program partitioning [36]. Most partitioning algorithms [21, 22] use program data. Based on sensitive program data, partition functions. These approaches' performance overhead depends on data marshalling expenses. Modern methods have 37-163% overhead [21, 22]. Unchecked code (or functions) should be segregated from checked code. We want the method to coexist with Checked C guarantees so code partitions keep spatial safety.

We propose type-directed code-centric program partitioning. Our system, CHECKEDCBOX, extends Checked c using **tainted (t\_\*)** types to denote isolated functions and pointers. The tainted types and Checked C allow pointer annotation along two dimensions: i) taintedness: a pointer can be either tainted or not (untainted), and (ii) checkedness: it can be either checked or not. The developer taints uncontrolled or hazardous routines and pointers. Second, CHECKEDCBOX divides the application into uc and c regions with distinct privileges:

* This partition only accesses tainted pointers and includes tainted types (functions and pointers).

*−*

*− −*

* c region (high-privilege unspoiled region): This partition holds the remaining (untainted) code and data. c-region functions can call uc-region functions, but not vice versa, except for call-back functions.

During program execution, the *uc* area partition will be executed in a sandboxed environment (e.g., WASM sandbox), and our compiler will provide the required instrumentation to permit communication between c and *uc* code.

Combining tainted types with privileged partitions ensures memory safety without marshalling overhead. c region functions should send uc region functions tainted pointers. To avoid marshalling, all tainted pointers are allocated in uc. Memory isolation prevents direct violations, but corrupted references can harm c area. Our compiler protects these attacks by ensuring tainted pointers only point to uc address space.