Tutorial: Making C Programs Safer with Checked C

Jie Zhou
University of Rochester
jzhou41@cs.rochester.edu

Michael Hicks
University of Maryland and
Correct Computation, Inc.
mwh@correctcomputation.com

Yudi Yang
University of Rochester
yyang116@u.rochester.edu

John Criswell
University of Rochester
criswell@cs.rochester.edu

***Abstract*—Despite its well-known lack of memory safety, C is still widely used to write both new code and to maintain legacy software. Extensive efforts to make C safe have not seen wide adoption due to poor performance and a lack of backward compatibility. Checked C is an open-source, safe extension to C that addresses these problems. This hands-on tutorial will introduce attendees to Checked C and provide guidance in the use of 3C, a semi-automatic tool that converts legacy C code to Checked C.**

I. Introduction

Many of the vulnerabilities discovered and fixed in production code arise from highly dangerous **violations of memory safety**; these include *buffer overflows*, *out-of-bounds reads*, and *use-after-frees*. Despite their more than 33-year history and the many attempts aimed at mitigating or blocking their exploitation, such vulnerabilities have remained a consistent, and sometimes worsening, threat [1], [2].

For new development, memory safety is not hard to attain: **memory safety vulnerabilities essentially occur in *only* C and C++ code** [3], so the requirement can be met by programming in *any other programming language*. Even for low-level programming, for which C and C++ used to be the only game in town, new languages like Rust and Go are viable, memory-safe options.

But what about the large number of active codebases already written in C and C++? It was for projects like these that Microsoft developed **Checked C** [4]–[6]. Checked C is an extension to C, comprising a set of types and annotations that, when used pervasively in a program, ensure its *spatial memory safety*, ruling out out-of-bounds reads and writes, null-pointer dereferences, and buffer overflows [4]. Previous work published at the SecDev conference has shown that Checked C is often unintrusive when retrofitting, and generates performant code [4], [7].

> In this tutorial, we will introduce attendees to Checked C, giving them a hands-on experience programming with it. We will also introduce attendees to **3C**, a tool that automatically annotates legacy C code with Checked C annotations. We show how 3C can be used to incrementally port a legacy project to Checked C so that it enjoys spatial memory safety.

Both Checked C and 3C are open source, and are built on Clang/LLVM [8], [9].

II. Background on Checked C and 3C

Checked C [4]–[6] extends C with support for *checked pointers*. To facilitate incremental porting, Checked C is *backward compatible* in the sense that all legacy C code is valid Checked C code. Moreover, legacy pointers may co-exist with checked pointers when following certain rules; e.g., you may not assign a legacy pointer to a checked pointer variable (without a special cast).

Checked C's *checked pointer types* include `ptr<`T`>` (ptr), `array_ptr<`T`>` (arr), and `nt_array_ptr<`T`>` ($ntarr$), which describe pointers to a single element, an array of elements, and a null-terminated array of elements of type T, respectively. Both arr and $ntarr$ pointers have an associated *bounds* which defines the range of memory referenced by the pointer. Here are the three different ways to specify the bounds for a pointer; the corresponding memory region is at the right.

`array_ptr<`T`> p: count(`n`)`	$[\mathtt{p}, \mathtt{p} + \mathtt{sizeof}(T) \times n)$
`array_ptr<`T`> p: byte_count(`b`)`	$[\mathtt{p}, \mathtt{p} + b)$
`array_ptr<`T`> p: bounds(`x, y`)`	$[x, y)$

The interpretation of an $ntarr$'s bounds is similar, but the range can extend further to the right, until a NULL terminator is reached (i.e., the NULL is not within the bounds).

Bounds expressions, like the n in `count(`n`)` above, may refer to in-scope variables that are neither modified nor have their address taken; bounds expressions must themselves be non-modifying. `struct` members can refer to adjacent fields in bounds expressions:

```
struct S { int len;
           array_ptr<int> buf: count(len); };
```

Checked C provides *interop types* (aka `itypes`) to allow legacy C functions to be given an *intended* checked type. Checked C supports enclosing code in special blocks called *checked regions*, which enforce more strict type checking rules that ensure within-region spatial safety [5], [6]. This property is particularly helpful to hunt for memory safety bugs in Checked C programs mixed with both checked pointers and original raw C pointers.

After typechecking, the Checked C compiler (which extends `clang`) instruments the program at dereferences (load or store) of checked pointers to check that (a) the pointer is not NULL and (b) that (if an arr or $ntarr$) the dereference is within the range of the declared bounds. Failed checks throw an exception. Oftentimes, inserted checks can be optimized away by LLVM [4], [7].

DOI 10.1109/SecDev51306.2021.00018

3C (which stands for *Checked C Converter*) is a tool that aims to assist a developer in porting a legacy C program to use Checked C. Completely porting arbitrary C to Checked C automatically is intractable—complicated code and unsafe uses will have no safe typing in Checked C. The 3C tool can effectively assist the human in the loop as they iteratively refactor their program, interspersing uses of the tool with manual changes.

3C employs two static analysis algorithms, called **typ3c** (pronounced "Types") and **boun3c** (pronounced "Bounce"). **typ3c** runs first to determine which legacy pointers can be made into checked pointers. **typ3c** leverages interop types to help it convert as many pointers as possible to their correct, checked type. For the pointers that remain, **typ3c** identifies the set of *root cause* code fragments, and lists them according to their magnitude of influence. This list indicates the areas the programmer should consider refactoring first.

After **typ3c** infers which pointers are checked, **boun3c** infers the needed bounds for `array_ptr<`T`>`, and `nt_array_ptr<`T`>` pointers, when possible. It works by correlating array pointers with potential bounds employed consistently at pointers' allocation and usage sites, connected by dataflow. **boun3c** is generally correct when it determines a bound, but will not determine bounds for every pointer.

Experience with 3C is promising: on 10 programs totaling 371K SLOC, 3C runs in a few tens of seconds, and automatically converts about 65% of the pointers to be checked, and about 75% of the bounds of those pointers that require them.

III. Tutorial Format

The tutorial will be broken down into three parts: (1) Background on Checked C; (2) Writing code in Checked C; and (3) Using 3C to port legacy C code.

The **first part** (Background) will present Checked C itself, in detail, including ongoing activities. We will discuss those programs that have been written in Checked C, and their performance. The **second part** will afford attendees the opportunity to write Checked C code from scratch. We will have small, curated exercises to build familiarity with Checked C features. Then we will have a somewhat larger, open-ended exercise that involves a bit more creativity. The **third part** will introduce attendees to 3C and how to use it to iteratively convert a legacy project. We will begin by walking attendees through a short tutorial (already up on Github). Then we will offer another program for attendees to port, on their own, and provide interactive feedback to help their process.

The tutorial will take 180 minutes in total, with three 10-minute breaks. We will preinstall all needed software on a Docker image of a Linux distribution which attendees can access during the tutorial.

IV. Expected Audience and Learning Outcomes

A. Expected Audience

Attendees should be familiar with C programming. They should have some familiarity with UNIX-style shell commands, e.g., to use `make` to build a project, or `clang` to compile an individual file. They should be familiar with Docker and have a machine with it installed so that they can run a Docker image on their own laptop or desktop.

B. Expected Learning Outcomes

First, the audience will learn about the basic design principles of Checked C. We hope the understanding of Checked C's most important design choices would facilitate its adoption.

Second, we expect the audience to get familiar with the core syntax of Checked C and be able to write simple Checked C programs with the checked pointers and bounds cast expressions.

Lastly, we expect the audience to learn how to use 3C to help them convert legacy C code to Checked C code. This includes running 3C, reading the output of it, and manually refactoring the parts that cannot be automatically converted by 3C but are provided refactoring suggestions given by 3C.

V. Personnel Qualifications

Michael Hicks is the CTO of Correct Computation, Inc (CCI) and a professor at the University of Maryland. He is a collaborator on Checked C since its early development; CCI is the technical owner of the 3C project. Jie Zhou is a doctoral student at the University of Rochester. He interned in the Checked C group at Microsoft in 2019 and has since been extending Checked C with temporal memory safety enforcement. John Criswell is Zhou's doctoral advisor and has collaborated with Zhou on the Checked C project. Yudi Yang is an undergraduate student at the University of Rochester who has extensive hands-on experience of refactoring Checked C code [7].

References

[1] C. Cimpanu, "Microsoft: 70 percent of all security bugs are memory safety issues," https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/, 2019.

[2] A. Gaynor, "What science can tell us about C and C++'s security," https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/, 2020, presentation at Enigma.

[3] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal War in Memory," in *Proc. of the IEEE Symposium on Security and Privacy*, 2013, pp. 48–62.

[4] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi, "Checked C: Making C Safe by Extension," in *2018 IEEE Cybersecurity Development (SecDev)*, Sep. 2018, pp. 53–60.

[5] A. Ruef, L. Lampropoulos, I. Sweet, D. Tarditi, and M. Hicks, "Achieving Safety Incrementally with Checked C," in *Principles of Security and Trust*, F. Nielson and D. Sands, Eds. Cham: Springer International Publishing, 2019, pp. 76–98.

[6] D. Tarditi, "Extending C with Bounds Safety and Improved Type Safety," Tech. Rep., 2021, version 0.9 of the Checked C specification. [Online]. Available: https://github.com/microsoft/checkedc/releases/tag/CheckedC-Clang-12.0.1-rel2

[7] J. Duan, Y. Yang, J. Zhou, and J. Criswell, "Refactoring the FreeBSD Kernel with Checked C," in *2020 IEEE Secure Development (SecDev)*, 2020, pp. 15–22.

[8] "The checked c clang repo." [Online]. Available: https://github.com/microsoft/checkedc-clang

[9] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO'04. Palo Alto, CA: IEEE Computer Society, 2004, pp. 75–86. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673