Formalizing and Evaluating Checked C

Michael Hicks
joint work with
Liyi Li, Yiyun Liu, Deena Postol, David Van Horn, and Leonidas Lampropoulos
University of Maryland
in consultation with the Checked C team at Microsoft

C/C++: Dangerous

- **Memory safety violations**, like HeartBleed [1], are the leading (and growing) cause of computer **security vulnerabilities** in software
  - 2019 Microsoft BlueHat report [2]: 70% of patches for memory safety bugs
  - 2019 MITRE report on CWE trends [3]: Buffer bounds errors the #1 most dangerous vulnerability, almost twice as dangerous as #2; the #5 error is buffer overreads
- The cause? Critical (inevitable) **defects in C/C++-based software**

[1] https://heartbleed.com

[2] https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf

[3] https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

C/C++: Not Going Away

- C/C++ software represents a huge, and growing footprint
  - 6.6 *billion* lines of C code as open source software [1]; another 1.7B of C++
  - 15% of monthly average users on Github are writing in C/C++, stable over past 5 years [2]
  - Customers increasingly want to put their legacy C/C++ systems code into networked environments (e.g., for Amazon and the FreeRTOS operating system)

- Porting legacy C/C++ code to a new language is expensive and risky
  - For new projects, using a new language makes sense
  - Rewriting existing code in a safe language would be time consuming and error prone
    - Languages like Rust, Haskell, Erlang, or Go are very different than C/C++
    - Rewriting very unlikely to be easy and fast

[1] https://www.openhub.net/languages/c

[2] https://www.benfrederickson.com/ranking-programming-languages-by-github-users/

Checked C: Spatially Safe C, Incrementally

- *Extends* C with **three new *checked* pointer types**
  - **Singleton** pointers **`_Ptr<T>`** — NULL or point to one ***T***
  - **Array** pointers **`_Array_ptr<T>` : `count(n)`** — NULL or point to an n-element buffer of ***T*** values (other ways to express bounds, too)
  - **Null-terminated array** pointers **`_NT_array_ptr<T>` : `count(n)`** — NULL or point to at least n values of type ***T***
- **Backward binary- and source- compatible** with legacy C
- Aims to achieve **spatial safety**: (1) use only checked pointers; (2) place in *checked regions*, which limit unsafe idioms. Pay as you go.

https://github.com/Microsoft/checkedc https://github.com/Microsoft/checkedc-clang

Strength of Safety Guarantee?

- Questions to consider:
  - **Is the Checked C design sound?** If programs adhere to its specification, are they indeed spatially safe?
  - What is the **impact** on spatial safety of the presence of **legacy code**?
  - Even if Checked C’s design is sound, there may be **bugs in the compiler**—how can these be avoided?
- Our approach to answering these questions:
  - **Develop a formal model**; prove properties about it
  - Use the formal model as the basis for **compiler validation**

Initial Work

- Formal model presented at POST 2019
- Proved **type safety** and **blame**
  - All safety violations can (in a formal sense) *blame* mixed-in legacy code
  - Mechanized proofs in the Coq proof assistant
- But the model was limited ("core"), lacking many important features
- No direct connection to the compiler

Achieving Safety Incrementally with Checked C

Andrew Ruef[1], Leonidas Lampropoulos[1,2], Ian Sweet[1], David Tarditi[3], and Michael Hicks[1]

[1] University of Maryland
{awruef,isw,llampro,mwh}@cs.umd.edu
[2] University of Pennsylvania
[3] Microsoft Research
dtarditi@microsoft.com

Abstract. Checked C is a new effort working toward a memory-safe C. Its design is distinguished from that of prior efforts by truly being an extension of C: Every C program is also a Checked C program. Thus, one may make incremental safety improvements to existing codebases while retaining backward compatibility. This paper makes two contributions. First, to help developers convert existing C code to use so-called *checked* (i.e., safe) pointers, we have developed a preliminary, automated porting tool. Notably, this tool takes advantage of the flexibility of Checked C's design: The tool need not perfectly classify *every* pointer, as required of prior all-or-nothing efforts. Rather, it can make a best effort to convert more pointers accurately, without letting inaccuracies inhibit compilation. However, such partial conversion raises the question: If safety violations can still occur, what sort of advantage does using Checked C provide? We draw inspiration from research on migratory typing to make our second contribution: We prove a *blame* property that renders so-called *checked regions* blameless of any run-time failure. We formalize this property for a core calculus and mechanize the proof in Coq.

1 Introduction

Vulnerabilities that compromise *memory safety* are at the heart of many attacks. *Spatial safety*, one aspect of memory safety, is ensured when any pointer dereference is always within the memory allocated to that pointer. *Buffer overruns* violate spatial safety, and still constitute a common cause of vulnerability. During 2012–2018, buffer overruns were the source of 9.7% to 18.4% of CVEs reported in the NIST vulnerability database [28], constituting the leading single cause of CVEs.

The source of memory unsafety starts with the language definitions of C and C++, which render out-of-bounds pointer dereferences "undefined." Traditional compilers assume they never happen. Many efforts over the last 20 years have aimed for greater assurance by proving that accesses are in bounds, and/or preventing out-of-bounds accesses from happening via inserted dynamic checks [26, 25, 30, 3, 15, 1, 2, 4, 7, 5, 8–10, 12, 6, 16, 22, 18]. This paper focuses on *Checked C*, a

This Work

- **Expanded the POST’19 model** to address many shortcomings
  - Mechanized in the Coq proof assistant
  - Implemented in PLT Redex
- Developed **randomized testing** framework
  - Based on the Redex model, and leverages its testing support
  - Used to compare code samples against the model and the actual compiler

Expanded Models

- **PLT Redex** and **Coq models** with many **more features**
  - dependent functions and function calls
  - dynamic (rather than static) array bounds
  - bounds expressions (to support pointer arithmetic)
  - null-terminated arrays, with *bounds widening*
  - dynamic bounds casts
- Theorems
  - **type safety** (basically, same as POST’19) — proved in Coq model
  - formal semantics **does not require “fat pointers”** to implement — stated and validated in PLT Redex

New Feature: Dynamically Sized Bounds

- **Dependent types** for **dynamically-sized** bounds

```
void foo(int c) {
  _Array_ptr<int> p: count(c) = malloc(c*sizeof(int));
}
```

  - Type `_Array_ptr<int> count(c)` *depends* on `c`, a run-time value
- Prior model could express static sizes; `_Array_ptr<int> count(5)`

New Feature: Bounds Expressions

- **Bounds expressions** support **pointer arithmetic**

```
  void foo(int c) {
    _Array_ptr<int> p: count(c) = malloc(c*sizeof(int));
    _Array_ptr<int> q: bounds(p,p+c) = p;
    q++;;
    *q = 1; // checks that p ≤ q < p+c
 }
```

- Prior model could support pointer arithmetic; only dynamic indexes (e.g., `p[1] = 1`, not `p++;*p = 1`)

New Feature: Null-terminated Arrays

- **Null-terminated Arrays** expand their bounds on non-null checks

```
void foo(_Nt_array_ptr<int> p) { // bounds(p,p)
  if (*p) { // expands to bounds(p,p+1)
    p[0] = ‘a’; // checks that p ≤ p < p+1
  }
  // bounds returned to bounds(p,p)
}
```

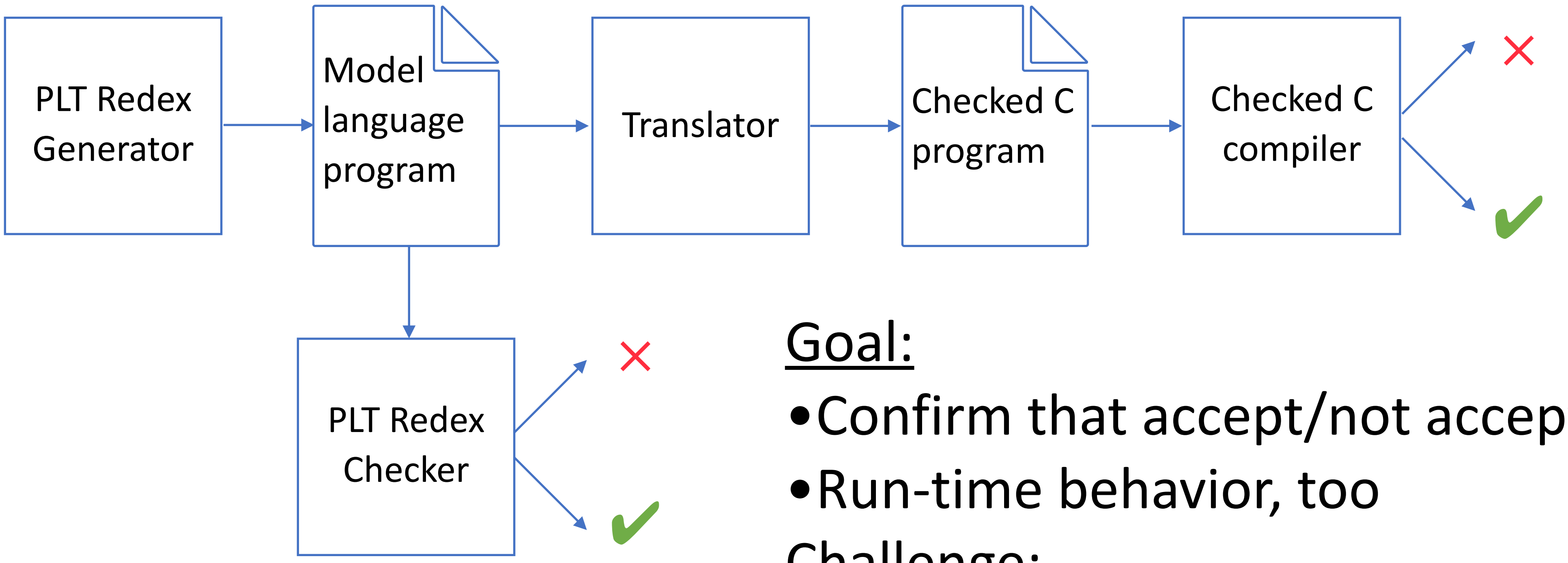
- Prior model had no support for null-terminated arrays and bounds widening

Proved Theorems

- **Type safety**: A program with only checked features (no legacy pointers) will not fail
  - By accessing undefined memory
  - By accessing an object contrary to its type
- **No fat pointers**: All Checked C pointers are single machine words
  - The formal semantics annotates pointers with their bounds; a direct translation would treat these annotations as “fat” metadata
  - Instead, we prove that a **type-driven transformation** can be run with a semantics without annotations, and is bisimilar to the original

Randomized, Model-based Testing

- A model is great. How to connect to the compiler? Randomized testing!



PLT Redex Generator

Model language program

Translator

Checked C program

Checked C compiler

✗

✓

PLT Redex Checker

✗

✓

<u>Goal:</u>

- Confirm that accept/not accept match
- Run-time behavior, too

<u>Challenge:</u>

- Producing diverse, interesting programs

Program Generation

- An arbitrary random program is unlikely to type check
    - Many more ill-formed abstract syntax trees than well formed ones
- Solution: Generate a **typing derivation**; *derive* program from it
    - Easier to generate well-formed derivations by construction
- Then: Produce an unsafe program by *mutating P*

Conclusions

- Checked C is a promising approach to securing legacy, and low-level code
    - But we want to ensure its design, implementation are solid
    - Our work is toward this goal

- Current status
    - Redex model is almost complete but requires some minor tweaks
    - Coq model has further to go, with some technical issues with dependent types and bounds widening still to solve
    - Key activity is automated test generation