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, constituting the leading single cause of CVEs.

In this talk we describe the current status of Checked C, a new effort working toward a memory-safe C. Checked C's 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. In addition, coding idioms not (yet) supported safely can still be accommodated, but require more scrutiny. Such scrutiny could be manual, or automated, i.e., as part of a static analysis that focuses less intently on memory-safe code.

Checked C permits intermixing checked (safe) pointers and legacy pointers. The former come in three varieties: pointers to single objects $\text{Ptr}(\tau)$; pointers to arrays $\text{Arrayptr}(\tau)$, and NUL-terminated arrays $\text{NtArrayptr}(\tau)$. The latter two have an associated clause that describes their known length in terms of constants and other program variables. The specified length is used to either prove pointer dereferences are safe or, barring that, serves as the basis of dynamic checks inserted by the compiler. Importantly, checked pointers are represented as in normal C--no changes to pointer structure (e.g., by “fattening” a pointer to include its bounds) are imposed. As such, interoperation with legacy C is eased. Moreover, the fact that checked and legacy pointers can be intermixed in the same module eases the porting process. Checked C provides an automated porting tool that takes advantage of the language’s flexibility.

An important question is what "safety" means in a program with a mix of checked and unchecked pointers. In such a program, safety violations are still possible. How, then, does one assess that a program is safer due to checking some, but not all, of its pointers? Unlike past safe-C efforts, Checked C specifically distinguishes parts of the program that are and may not be fully "safe." So-called checked regions differ from unchecked ones in that they can only use checked pointers--dereference or creation of unchecked pointers, unsafe casts, and other potentially dangerous constructs are disallowed. Using a core calculus for Checked C programs called CoreChkC, we have proved that these restrictions are sufficient to ensure that checked code cannot be blamed. That is, checked code is internally safe, and any run-time failure can be attributed to unchecked code, even if that failure occurs in a checked region. This proof has been fully formalized in the Coq proof assistant.

Checked C is implemented as an extension to the Clang/ LLVM compiler. The compiler inserts run-time checks for the evaluation of lvalue expressions whose results are derived from checked pointers and that will be used to access memory. Accessing a $\text{Ptr}(\tau)$ requires a null check, while accessing an $\text{Arrayptr}(\tau)$ requires both null and bounds checks. The code for these checks is handed to LLVM, which we allow to remove checks if it can prove they will always pass. In general, such checks are the only source of Checked C run-time overhead. Preliminary experiments on some small, pointer-intensive benchmarks show running time overhead to be around 8.6%, on average. Checked C’s development is active and ongoing. Current activities include further development of the language’s type system; improved automated porting support; and more sophisticated optimizations.

Checked C is hosted on Github at https://github.com/Microsoft/checkedc, and we welcome collaborators.
References:

**Michael W. Hicks** is a Professor in the Computer Science department and the CTO of Correct Computation, Inc. He recently completed a three-year term as Chair of ACM SIGPLAN, the Special Interest Group in Programming Languages, and was the first Director of the University of Maryland's Cybersecurity Center (MC2). His research focuses on using programming languages and analyses to improve the security, reliability, and availability of software. He has explored the design of new programming languages and analysis tools for helping programmers find bugs and software vulnerabilities, and explored technologies to shorten patch application times by allowing software upgrades without downtime. Recently he has been looking at synergies between cryptography and programming languages, as well techniques involving random testing and probabilistic reasoning. He also led the development of a new security-oriented programming contest, "build-it, break-it, fix-it," which has been offered to the public and to students of his Coursera class on Software Security. He blogs at [http://www.pl-enthusiast.net/](http://www.pl-enthusiast.net/).

---

**License:** Creative Commons 2.5

---

Other available formats:

**Checked C: Safe C, Incrementally**

---

**Academia Presentation HCSS 2019**