Memory and Type Safety for Embedded Systems

Paul Soulier
psoulier@hawaii.edu
University of Hawaii, Manoa

Depeng Li
depengli@hawaii.edu
University of Hawaii, Manoa

Abstract

Security vulnerabilities that originate from memory-related programming errors are all too common in systems programs and embedded systems with numerous such flaws being reported each year. Over the past several decades, many application domains have enjoyed advances in programming languages that make these variety of errors all but non-existant while the vast majority of embedded systems are still written in the C programming language [3, 1]. The reasons for the continued use of C are certainly numerous, but there is one aspect of the C languages that sets it apart from other languages: power of expression. This power of expression enables programmers to write powerful applications for even the most obscure and obtuse hardware platforms that would not be possible in other languages; but it comes at a cost. The C language is neither type safe nor memory safe. The goal of this research is to define a type system that can ultimately be used in an alternative language to C. The type system must be type safe, memory safe, and reduce unexpected state modifications due to pointer aliasing and concurrent execution while still maintaining the expressive power that is found in C.

Type and memory safe systems are well understood with many existing languages implementing them. However, embedded systems pose unique challenges to type systems that are not addressed by most languages where the necessity to interact with hardware through memory-mapped register windows or through protocols necessitates the ability to precisely define and manipulate data structures. For example, to ensure out-of-bounds access is not permitted, arrays in many languages maintain the number of elements contained within by adding to the overall memory required by the array. A protocol that utilizes an array defines only the space required by the elements. The type system must provide mechanisms for ensuring such data structures maintain memory-safe operations without altering the in-memory data organization.

Multi-processor devices have become commonplace. With the proliferation of multiprocessor architectures, the complexity of maintaining memory integrity also increases. Language-level support for concurrency offers many advantages in providing memory-safe operations. Software-based isolation (as found in the Singularity project [2]) can confine memory access to within a given process and ensure accesses outside of the process employ proper IPC mechanisms. This isolation mechanism is enforced through the language type system providing the significant advantage over hardware MMUs of detecting errors at compile time rather than runtime. Furthermore, various embedded system platforms do not contain hardware-based MMUs making this the only viable option to provide any memory isolation.

Aliasing in a software system occurs when a single object in memory is addressed by more than one reference. Aliasing has the potential to reduce the comprehensibility of code that can, in turn, lead to unanticipated modifications to data when objects are manipulated in unexpected ways. References are generally used to pass an object from one procedure or object to another such that it can be modified. Frequently, there is little need to maintain multiple references to a single object. The use of value semantics and linear types can help minimize the occurrence of aliased objects thus reducing the negative potential issues that arise from aliased objects. Linear types also have the potential to offer a efficient message-based IPC mechanism.

With the ability incorporate more computational power into smaller packages and wireless connectivity, embedded systems are finding their way into many critical applications such as insulin pumps, sensor networks, and industrial control devices. Given the potential consequences of security flaws in such devices the need for new type systems and programming languages to prevent software flaws is a critical component to ensuring the security of future embedded software systems.
References

