On Software Safety, Security, and Abstract Interpretation

Daniel Kästner, Laurent Mauborgne, Christian Ferdinand

AbsInt Angewandte Informatik GmbH

Abstract. Static code analysis can be applied to show compliance to coding guidelines, and to demonstrate the absence of critical programming errors, including runtime errors and data races. In recent years, security concerns have become more and more relevant for safety-critical systems, not least due to the increasing importance of highly-automated driving and pervasive connectivity. While in the past, sound static analyzers have been primarily applied to demonstrate classical safety properties they are well suited also to address data safety, and to discover security vulnerabilities. This article gives an overview and discusses practical experience.

1 Introduction

In safety-critical systems, static analysis plays an important role. With the growing size of software-implemented functionality, preventing software-induced system failures becomes an increasingly important task. One particularly dangerous class of errors are runtime errors which include faulty pointer manipulations, numerical errors such as arithmetic overflows and division by zero, data races, and synchronization errors in concurrent software. Such errors can cause software crashes, invalidate separation mechanisms in mixed-criticality software, and are a frequent cause of errors in concurrent and multi-core applications. At the same time, these defects are also at the root of many security vulnerabilities, including exploits based on buffer overflows, dangling pointers, or integer errors.

In safety-critical software projects, obeying coding guidelines such as MISRA C is strongly recommended by safety standards like DO-178C, IEC-61508, ISO-26262, or EN-50128. In addition, all of them consider demonstrating the absence of runtime errors explicitly as a verification goal. This is often formulated indirectly by addressing runtime errors (e.g., division by zero, invalid pointer accesses, arithmetic overflows) in general, and additionally considering corruption of content, synchronization mechanisms, and freedom of interference in concurrent execution. Semantics-based static analysis has become the predominant technology to detect runtime errors and data races.

Abstract interpretation-based static analyzers provide full control and data coverage and allow conclusions to be drawn that are valid for all program runs with all inputs. Such conclusions may be that no timing or space constraints are

violated, or that runtime errors or data races are absent: the absence of these errors can be guaranteed.

In the past, security properties have mostly been relevant for non-embedded and/or non-safety-critical programs. Recently due to increasing connectivity requirements (cloud-based services, car-to-car communication, over-the-air updates, etc.), more and more security issues are rising in safety-critical software as well.

Safety-critical software is developed according to strict guidelines which improve software verifiability. As an example dynamic memory allocation and recursion often are forbidden or used in a very limited way. Stronger code properties can be shown, so that also security vulnerabilities can be addressed in a more powerful way.

The topic of this article is to show that some classes of defects can be proven to be absent in the software so that exploits based on such defects can be excluded. Additional syntactic checks and semantical analyses become necessary to address security properties that are orthogonal to safety requirements.

2 Security in Safety-Critical Systems

MISRA C aims at avoiding programming errors and enforcing a programming style that enables the safest possible use of C. A particular focus is on dealing with undefined/unspecified behavior of C and on preventing runtime errors. As a consequence, it is also directly applicable to security-relevant code, which is explicitly addressed by Amendment 1 to MISRA C:2012. Other well-known coding guidelines are the ISO/IEC TS 17961, SEI CERT C, and the MITRE Common Weakness Enumeration CWE. The most prominent vulnerabilities at the C code level which are addressed in all coding guidelines are the following: Stack-based buffer overflows, heap-based buffer overflows, general invalid pointer accesses, uninitialized memory accesses , integer errors, format string vulnerabilities, and concurrency defects.

Most of these vulnerabilities are based on undefined behaviors, and among them buffer overflows seem to play the most prominent role. Most of them can be used for denial-of-service attacks by crashing the program or causing erroneous behavior. They can also be exploited to inject code and cause the program to execute it, and to extract confidential data from the system. It is worth noticing that from the perspective of a static analyzer most exploits are based on potential runtime errors: when using an unchecked value as an index in an array the error will only occur if the attacker manages to provide an invalid index value. The obvious conclusion is that safely eliminating all potential runtime errors due to undefined behaviors in the program significantly reduces the risk for security vulnerabilities.

From a semantical point of view, a safety property can always be expressed as a trace property. This means that to find all safety issues, it is enough to look at each trace of execution in isolation.

This is not possible any more for security properties. Most of them can only be expressed as set of traces properties, or hyperproperties [2]. A typical example is non-interference [7]: to express that the final value of a variable x can only be affected by the initial value of y and no other variable, one must consider each pair of possible execution traces with the same initial value for y, and check that the final value of x is the same for both executions. It was proven in [2] that any other definition (tracking assignments, etc) considering only one execution trace at a time would miss some cases or add false dependencies. This additional level of sets has direct consequences on the difficulty to track security properties soundly.

Finding expressive and efficient abstractions for such properties is a young research field (see [1]), which is the reason why no sound analysis of such properties appear in industrial static analyzers yet. The best solution using the current state of the art consists of using dedicated safety properties as an approximation of the security property in question, such as the taint propagation described below.

3 Proving the Absence of Defects

In the following we will concentrate on the sound static runtime error analyzer Astrée [3]. It reports program defects caused by unspecified and undefined behaviors and program defects caused by invalid concurrent behavior. Users are notified about: integer/floating-point division by zero, out-of-bounds array indexing, erroneous pointer manipulation and dereferencing (buffer overflows, null pointer dereferencing, dangling pointers, etc.), data races, lock/unlock problems, deadlocks, integer and floating-point arithmetic overflows, read accesses to uninitialized variables, unreachable code, non-terminating loops, and violations of coding rules (MISRA C, ISO/IEC TS 17961, CERT, CWE).

Astrée computes data and control flow reports containing a detailed listing of accesses to global and static variables sorted by functions, variables, and processes and containing a summary of caller/called relationships between functions. The analyzer can also report each effectively shared variable, the list of processes accessing it, and the types of the accesses (read, write, read/write).

To deal with concurrency defects, Astrée implements a sound low-level concurrent semantics [5] which provides a scalable sound abstraction covering all possible thread interleavings. In addition to the classes of runtime errors found in sequential programs, Astrée can report data races, and lock/unlock problems, i.e., inconsistent synchronization. After a data race, the analysis continues by considering the values stemming from all interleavings. Since Astrée is aware of all locks held for every program point in each concurrent thread, Astrée can also report all potential deadlocks. Practical experience on avionics and automotive industry applications are given in [3][6].

Sophisticated data and control flow information can be provided by two dedicated analysis methods: program slicing and taint analysis. Program slicing aims

at identifying the part of the program that can influence a given set of variables at a given program point. Applied to a result value, e.g., it shows which functions, which statements, and which input variables contribute to its computation. Taint analysis tracks the propagation of specific data values through program execution. It can be used, e.g., to determine program parts affected by corrupted data from an insecure source. A sound taint analyzer will compute an over-approximation of the memory locations that may be mapped to a tainted value during program execution [4].

4 Conclusion

In this article, we have listed code-level defects and vulnerabilities relevant for functional safety and security. We have shown that many security attacks can be traced back to behaviors undefined or unspecified according to the C semantics. By applying sound static runtime error analyzers, a high degree of security can be achieved for safety-critical software. Security hyperproperties require additional analyses to be performed which, by nature, have a high complexity. We have given two examples of scalable dedicated analyses, program slicing and taint analysis. Applied as extensions of sound static analyzers, they allow to further increase confidence in the security of safety-critical embedded systems.

Acknowledgment: This work was funded within the project ARAMiS II by the German Federal Ministry for Education and Research with the funding ID 01—S16025. The responsibility for the content remains with the authors.

References

1. Assaf, M., Naumann, D.A., Signoles, J., Totel, E., Tronel, F.: Hypercollecting semantics and its application to static analysis of information flow. CoRR abs/1608.01654 (2016), http://arxiv.org/abs/1608.01654 [retrieved: Sep. 2017].
2. Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security 18, 1157–1210 (2010)
3. Kästner, D., Miné, A., Mauborgne, L., Rival, X., Feret, J., Cousot, P., Schmidt, A., Hille, H., Wilhelm, S., Ferdinand, C.: Finding All Potential Runtime Errors and Data Races in Automotive Software. In: SAE World Congress 2017. SAE International (2017)
4. Kästner, D., Mauborgne, L., Ferdinand, C.: Detecting Safety- and Security-Relevant Programming Defects by Sound Static Analysis. In: Rainer Falk, Steve Chan, J.C.B. (ed.) The Second International Conference on Cyber-Technologies and Cyber-Systems (CYBER 2017). IARIA Conferences, vol. 2, pp. 26–31. IARIA XPS Press (2017)
5. Miné, A.: Static analysis of run-time errors in embedded real-time parallel C programs. Logical Methods in Computer Science (LMCS) 8(26), 63 (Mar 2012)
6. Miné, A., Delmas, D.: Towards an Industrial Use of Sound Static Analysis for the Verification of Concurrent Embedded Avionics Software. In: Proc. of the 15th International Conference on Embedded Software (EMSOFT'15). pp. 65–74. IEEE CS Press (Oct 2015)

7. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)