Advanced Sound Static Analysis to Detect Safety- and Security-Relevant Programming Defects

Daniel Kästner, Laurent Mauborgne, Nicolas Grafe, Christian Ferdinand

AbsInt GmbH
Science Park 1, 66123 Saarbrücken, Germany
Email: kaestner@absint.com, mauborgne@absint.com, grafe@absint.com, ferdinand@absint.com

Abstract—**Static code analysis has evolved to be a standard technique in the development process of safety-critical software. It 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.**

***Keywords*–static analysis; abstract interpretation; runtime errors; security vulnerabilities; functional safety; cybersecurity.**

I. INTRODUCTION

Some years ago, static analysis meant manual review of programs. Nowadays, automatic static analysis tools are gaining popularity in software development as they offer a tremendous increase in productivity by automatically checking the code under a wide range of criteria [1]. Many software development projects are developed according to coding guidelines, such as MISRA C [2], SEI CERT C [3], or CWE (Common Weakness Enumeration) [4], aiming at a programming style that improves clarity and reduces the risk of introducing bugs. Compliance checking by static analysis tools has become common practice.

In safety-critical systems, static analysis plays a particularly important role. A failure of a safety-critical system may cause high costs or even endanger human beings. 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 all current safety standards, including DO-178C [5], IEC-61508 [6], ISO-26262 [7], and EN-50128 [8]. 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 is a formal methodology for semantics-based static program analysis [9]. It supports formal soundness proofs (it can be proven that no error is missed) and scales to real-life industry applications. 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 [10]. Nowadays, abstract interpretation-based static analyzers that can detect stack overflows and violations of timing constraints [11] and that can prove the absence of runtime errors and data races [12][13], are widely used for developing and verifying safety-critical software. From a methodological point of view, abstract interpretation-based static analyses can be seen as equivalent to testing with full data and control coverage. They do not require access to the physical target hardware, can be easily integrated in continuous verification frameworks and model-based development environments [14], and they allow developers to detect runtime errors as well as timing and space bugs in early product stages.

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. Security exploits like the Jeep Cherokee hacks [15], which affect the safety of the system, are becoming more and more frequent. In consequence, safety-critical software development faces novel challenges, which previously only have been addressed in other industry domains.

On the other hand, as outlined above, safety-critical software is developed according to strict guidelines, which effectively reduce the relevant subset of the programming language used and improve software verifiability. As an example dynamic memory allocation and recursion often are forbidden or used in a very limited way. In consequence, for safety-critical software much stronger code properties can be shown than for non-safety-critical software, 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. On the other hand, additional syntactic checks and semantical analyses become necessary to address security properties that are orthogonal to safety requirements. Throughout the article we will focus on software aspects only, without addressing safety or security properties at the hardware level. While we focus on the programming language C, the basic analysis techniques described in this article are applicable to other programming languages as well.

The article is based on [1], amplifying some of the key aspects and following up on ongoing work. It is structured as follows: Section II discusses the relation between *safety* and *security* requirements. After a brief summary of typical requirements and verification goals formulated in current safety standards the role of coding standards is discussed in Section II-B. A classification of security vulnerabilities is given in Section II-C, and Section II-D focuses on the analysis complexity of safety and security properties. Section III gives an overview of abstract interpretation and its application to runtime error analysis, using the sound analyzer Astrée as an example. Section IV gives an overview of control and data flow analysis with emphasis on two advanced analysis techniques: program slicing (cf. Section IV-A) and taint analysis (cf. Section IV-B). Section V concludes.

II. SECURITY IN SAFETY-CRITICAL SYSTEMS

Functional safety and security are aspects of dependability, in addition to reliability and availability. *Functional safety* is usually defined as the absence of unreasonable risk to life and property caused by malfunctioning behavior of the software. The main goals of *information security* or *cybersecurity* (for brevity denoted as "*security*" in this article) traditionally are to preserve *confidentiality* (information must not be disclosed to unauthorized entities), *integrity* (data must not be modified in an unauthorized or undetected way), and *availability* (data must be accessible and usable upon demand).

In safety-critical systems, safety and security properties are intertwined. A violation of security properties can endanger the functional safety of the system: an information leak could provide the basis for a successful attack on the system, and a malicious data corruption or denial-of-service attack may cause the system to malfunction. Vice versa, a violation of safety goals can compromise security: buffer overflows belong to the class of critical runtime errors whose absence have to be demonstrated in safety-critical systems. At the same time, an undetected buffer overflow is one of the main security vulnerabilities, which can be exploited to read unauthorized information, to inject code, or to cause the system to crash [16]. To emphasize this, in a safety-critical system the definition of functional safety can be adapted to define cybersecurity as absence of unreasonable risk to life and property caused by malicious misuse of the software.

The convergence of safety and security properties also becomes apparent in the increasing role of data in safety-critical systems. There are many well-documented incidents where harm was caused by erroneous data, corrupted data, or inappropriate use of data – examples include the Turkish Airlines A330 incident (2015), the Mars Climate Orbiter crash (1999), or the Cedars Sinai Medical Centre CT scanner radiation overdose (2009) [17]. The reliance on data in safety-critical systems has significantly grown in the past few years, cf. e.g., data used for decision-support systems, data used in sensor fusion for highly automatic driving, or data provided by car-to-car communication or downloaded from a cloud. As a consequence of this there are ongoing activities to provide specific guidance for handling data in safety-critical systems [17]. At the same time, these data also represent safety-relevant targets for security attacks.

In this section we will first give an overview of typical verification goals and requirements of contemporary safety norms, followed by a brief discussion of relevant coding guidelines. With this background we will present a classification of security vulnerabilities and discuss their relationship with respect to safety requirements. The section concludes with a discussion of algorithmic complexity of safety and security requirements.

A. The Safety Standards' Perspective

Safety standards like ISO-26262 [7], DO-178B [18], DO-178C [5], IEC-61508 [6], and EN-50128 [8] require to identify functional and non-functional hazards and to demonstrate that the software does not violate the relevant safety goals. Some non-functional safety hazards can be critical for the correct functioning of the system: violations of timing constraints in real-time software and software defects like runtime errors or stack overflows. Depending on the criticality level of the software the absence of safety hazards has to be demonstrated by formal methods or testing with sufficient coverage. In this section we will focus on the non-functional aspects at the programming language level, and use the avionics standard DO178C and the automotive standard ISO-26262 as examples.

1) DO-178C: Published in 2011, the DO-178C [5] ("Software Considerations in Airborne Systems and Equipment Certification"), is the primary document by which certification authorities such as EASA or FAA approve commercial software-based aerospace systems. In general, the DO-178C aims at providing "guidance for determining, in a consistent manner and with an acceptable level of confidence, that the software aspects of airborne systems and equipment comply with airworthiness requirements." The *Formal Methods Supplement* DO-333 [19] gives an overview of formal methods, such as Abstract Interpretation (cf. Section III-A), and their application in the software development and verification process.

In the software development process, *Software Design Standards* and *Software Code Standards* have to be taken into account to "disallow the use of constructs or methods that produce outputs that cannot be verified or that are not compatible with safety-related requirements". The *Software Design Standards* (cf. Section 11.7 of [5]) are defined to focus on algorithmic constraints like exclusion of recursion, dynamic objects, or data aliases. They should also include complexity restrictions like maximum level of nested calls. The *Software Code Standards* focus on the programming language. They identify the programming language to be used and should define a safety-oriented language subset (cf. Section 11.8a of [5]). Coding guidelines enforcing compliance with the *Software Design Standard* and the *Software Code Standards* have to be applied.

One objective of the software verification activities is the *accuracy and consistency* verification goal, which aims at determining "the correctness and consistency of the source code, including stack usage, memory usage, fixed point arithmetic overflow and resolution, [...], worst-case execution timing, exception handling, use of uninitialized variables, [...] and data corruption due to task or interrupt conflicts". In particular, this includes runtime errors caused by undefined or unspecified behavior of the programming language. Runtime errors also have to be addressed when verifying the software component integration, implying, e.g., detecting incorrect initialization of variables, parameter passing errors, and data corruption. All these requirements can be checked using formal analysis (cf. Section FM.6.3.4 of [19]). Furthermore, data and control flow analysis is required at the software architectural level and the source code level to ensure implementation consistency.

2) ISO-26262: In this section we focus on Part 6 of the ISO-26262 standard [20], which specifies the requirements for product development at the software level for automotive applications.

Supporting real-time software and runtime error handling is considered a basic requirement for selecting a suitable modeling or programming language (cf. Section 5.4.6 of [20]). It also states that "criteria that are not sufficiently addressed by the language itself shall be covered by the corresponding guidelines or by the development environment" (cf. Section 5.4.7 of [20]). Coding guidelines have to be applied to support the correctness of the design and implementation. For the programming language C the MISRA C and MISRA AC AGC standards are suggested. The goals to be addressed include "the use of language subsets to exclude language constructs which could result in unhandled runtime errors". The absence of runtime errors has to be ensured by appropriate tools as a part of the development environment.

Among the verification goals of the software design and implementation stage are correctness of data flow and control flow between and within the software units, consistency of the interfaces, and robustness. The standard lists some examples for robustness properties, including implausible values, execution errors, division by zero, and errors in the data and control flow. Furthermore, the compatibility of the software unit implementation with the target hardware has to be ensured.

The software integration phase considers functional dependencies and the dependencies between software integration and hardware-software integration, including non-functional software properties. Again robustness properties have to be taken into account, and it has to be ensured that sufficient resources are available.

The methods for verification of software unit and design and the methods for software verification of software integration include static analysis in general and abstract interpretation (cf. [21]).

B. Coding Guidelines

The MISRA C standard [2] has originally been developed with a focus on automotive industry but is now widely recognized as a coding guideline for safety-critical systems in general. Its aim is to avoid programming errors and enforce 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.

The most prominent coding guidelines targeting security aspects are the ISO/IEC TS 17961 [22], the SEI CERT C Coding Standard [3], and the MITRE Common Weakness Enumeration CWE [4].

The ISO/IEC TS 17961 C Secure Coding Rules [22] specifies rules for secure coding in C. It does not primarily address developers but rather aims at establishing requirements for compilers and static analyzers. MISRA C:2012 Addendum 2 [23] compares the ISO/IEC TS 17961 rule set with MISRA C:2012. Only 4 of the C Secure rules are not covered by the first edition of MISRA C:2012 [2], however, with Amendment 1 to MISRA C:2012 [24] all of them are covered as well. This illustrates the strong overlap between the safety- and security-oriented coding guidelines.

The SEI CERT C Coding Standard belongs to the CERT Secure Coding Standards [25]. While emphasizing the security aspect CERT C [3] also targets safety-critical systems: it aims at "developing safe, reliable and secure systems". CERT distinguishes between rules and recommendations where rules are meant to provide normative requirements and recommendations are meant to provide general guidance; the book version [3] describes the rules only. A particular focus is on eliminating undefined behaviors that can lead to exploitable vulnerabilities. In fact, almost half of the CERT rules (43 of 99 rules) are targeting undefined behaviors according to the C norm. Addendum 3 to MISRA C:2012 [26] provides an in-depth analysis of the coverage of MISRA C:2012 against CERT C [3].

The Common Weakness Enumeration CWE is a software community project [4] that aims at creating a catalog of software weaknesses and vulnerabilities. The goal of the project is to better understand flaws in software and to create automated tools that can be used to identify, fix, and prevent those flaws. There are several catalogues for different programming languages, including C. In the latter one, once again, many rules are associated with undefined or unspecified behaviors.

C. Vulnerability Classification

Many rules are shared between the different coding guidelines, but there is no common structuring of security vulnerabilities. The CERT Secure C roughly structures its rules according to language elements, whereas ISO/IEC TS 17961 and CWE are structured as a flat list of vulnerabilities. In the following we list some of the most prominent vulnerabilities, which are addressed in all coding guidelines and which belong to the most critical ones at the C programming level. The presentation follows the overview given in [16].

1) Stack-based Buffer Overflows: An array declared as local variable in C is stored on the runtime stack. A C program may write beyond the end of the array due to index values being too large or negative, or due to invalid increments of pointers pointing into the array. A runtime error then has occurred whose behavior is undefined according to the C semantics. As a consequence the program might crash with bus error or segmentation fault, but typically adjacent memory regions will be overwritten. An attacker can exploit this by manipulating the return address or the frame pointer both of which are stored on the stack, or by indirect pointer overwriting, and thereby gaining control over the execution flow of the program. In the first case the program will jump to code injected by the attacker into the overwritten buffer instead of executing an intended function return. In case of overflows on array read accesses confidential information stored on the stack (e.g., through temporary local variables) might be leaked. An example of such an exploit is the well-known W32.Blaster.Worm [27].

In the following we will illustrate this vulnerability and its implications for safety and security with a small toy example.

Consider the following C code fragment:

```
struct {
  char buf[4096];
  unsigned int len;
} msg;

int f1(void) {
  char LBuf [1024];
  unsigned int i;
  GetMsg();
  for (i=0; i<msg.len; i++)
    LBuf[i]=msg.buf[i];
}

void f0(void) {
  ...
  f1(a,b);
  ...
}
```

Function `f0` calls function `f1`, which has two local variables, a `char` array `LBuf` of size 1024 bytes, and an integer `i`. Let's assume the function `GetMsg` updates the global variable `msg` from the environment or a user input. The code expects `msg` to contain a valid string in field `buf`, and the length of this string in field `len`.

There is no explicit check that the value of `len` is not greater than 1024. If `len>1024` a buffer overwrite will occur, and contents of the runtime stack will be overwritten.

A possible stack layout before executing the loop in function `f1` is shown in Figure 1. We assume that the stack grows downwards. The stack frame of `f0` starts with its return address, the previous value of the frame pointer, space for local variables and possibly spilled registers. It is followed by the stack frame of `f1`. Its return address is the instruction following the call instruction in the code of `f0` from which `f1` was invoked. The next cell points to the previous frame pointer, then there are 1024 bytes for the local array `LBuf`, and 4 bytes for the local variable `i`.

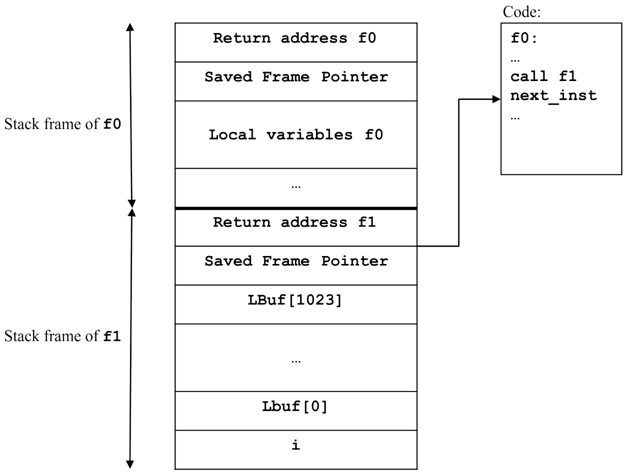


Figure 1. Stack frame before start of loop in `f1`

The loop in `f1` overwrites all elements of the `LBuf` array, starting with index 0. When a buffer overflow happens, the overwriting does not stop in the cell of `LBuf[1023]`, instead the entries of the stack above it will be overwritten, starting with the previous frame pointer, the return address, etc. The expected consequence will be that the program will behave erratically or crash, when trying to continue execution at a wrong (overwritten) return address.

When the situation is exploited in a malicious way, the return address is intentionally overwritten with a carefully selected address. The stack frame of such a code injection attack is shown in Figure 2.

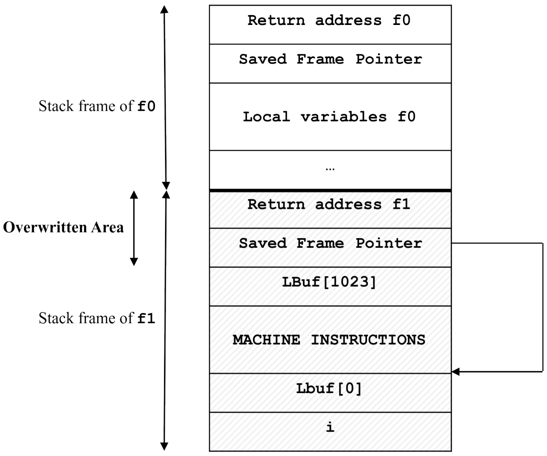


Figure 2. Stack frame after code injection attack

The assumption in that case is that the attacker has gained control to influence the value of the fields in `msg`. The first step is to fill a part of the entries of the source buffer with bytes representing feasible machine code. The `len` parameter is chosen such that the two stack cells above `LBuf` are overwritten. In the second step, the attacker makes sure that the entries in the `msg.buf` array, which will be written in the return address field of the stack, contain the address within `LBuf` where the injected code begins. Then function `f1` will continue normally, but instead of returning to its caller, it will execute the injected code.

2) Heap-based Buffer Overflows: Heap memory is dynamically allocated at runtime, e.g., by calling *malloc*() or *calloc*() implementations provided by dynamic memory allocation libraries. There may be read or write operations to dynamically allocated arrays that access beyond the array boundaries, similarly to stack-allocated arrays. In case of a read access information stored on the heap might be leaked – a prominent example is the Heartbleed bug in OpenSSL (cf. CERT vulnerability 720951 [28]). Via write operations attackers may inject code and gain control over program execution, e.g., by overwriting management information of the dynamic memory allocator stored in the accessed memory chunk.

3) General Invalid Pointer Accesses: Buffer overflows are special cases of invalid pointer accesses, which are listed here as separate points due to the large number of attacks based on them. However, any invalid pointer access in general is a security vulnerability – other examples are null pointer accesses or dangling pointer accesses. Accessing such a pointer is undefined behavior, which can cause the program to crash, or behave erratically. A dangling pointer points to a memory location that has been deallocated either implicitly (e.g., data stored in the stack frame of a function after its return) or explicitly by the programmer. A concrete example of a dangling pointer access is the double free vulnerability where already freed memory is freed a second time. This can be exploited by an attacker to overwrite arbitrary memory locations and execute injected code [16].

4) Uninitialized Memory Accesses: Automatic variables and dynamically allocated memory have indeterminate values when not explicitly initialized. Accessing them is undefined behavior. Uninitialized variables can also be used for security attacks, e.g., in CVE-2009-1888 [29] potentially uninitialized variables passed to a function were exploited to bypass the access control list and gain access to protected files [3].

5) Integer Errors: Integer errors are not exploitable vulnerabilities by themselves, but they can be the cause of critical vulnerabilities like stack- or heap-based buffer overflows. Examples of integer errors are arithmetic overflows, or invalid cast operations. If, e.g., a negative signed value is used as an argument to a $memcpy()$ call, it will be interpreted as a large unsigned value, potentially resulting in a buffer overflow.

6) Format String Vulnerabilities : A format string is copied to the output stream with occurrences of %-commands representing arguments to be popped from the stack and expanded into the stream. A format string vulnerability occurs, if attackers can supply the format string because it enables them to manipulate the stack, once again making the program write to arbitrary memory locations.

7) Concurrency Defects: Concurrency errors may lead to concurrency attacks, which allow attackers to violate confidentiality, integrity and availability of systems [30]. In a race condition the program behavior depends on the timing of thread executions. A special case is a write-write or read-write data race where the same shared variable is accessed by concurrent threads without proper synchronization. In a Time-of-Check-to-Time-of-Use (TOCTOU) race condition the checking of a condition and its use are not protected by a critical section. This can be exploited by an attacker, e.g., by changing the file handle between the accessibility check and the actual file access. In general, attacks can be run either by creating a data race due to missing lock-/unlock protections, or by exploiting existing data races, e.g., by triggering thread invocations.

Most of the vulnerabilities described above are based on undefined behaviors, and among them buffer overflows seem to play the most prominent role for real-live attacks. 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.

D. Analysis Complexity

While semantics-based static program analysis is widely used for safety properties, there is practically no such analyzer dedicated to specific security properties. This is mostly explained by the difference in complexity between safety and security properties. 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 [31]. A typical example is non-interference [32]: 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 [31] 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.

Other examples of hyperproperties are secure information flow policies, service level agreements (which describe acceptable availability of resources in term of mean response time or percentage uptime), observational determinism (whether a system appears deterministic to a low-level user), or quantitative information flow.

Finding expressive and efficient abstractions for such properties is a young research field (see [33]), 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 in Section IV-B.

III. PROVING THE ABSENCE OF DEFECTS

In safety-critical systems, the use of dynamic memory allocation and recursions typically is forbidden or only used in limited ways. This simplifies the task of static analysis such that for safety-critical embedded systems it is possible to formally prove the absence of runtime errors, or report all potential runtime errors which still exist in the program. Such analyzers are based on the theory of abstract interpretation [9], a mathematically rigorous formalism providing a semantics-based methodology for static program analysis.

A. Abstract Interpretation

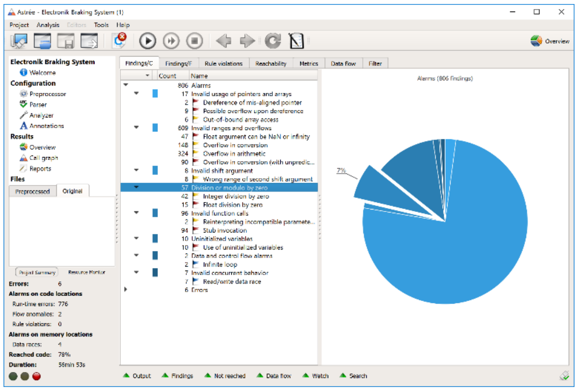
The semantics of a programming language is a formal description of the behavior of programs. The most precise semantics is the so-called concrete semantics, describing closely the actual execution of the program on all possible inputs. Yet in general, the concrete semantics is not computable. Even under the assumption that the program terminates, it is too detailed to allow for efficient computations. The solution is to introduce an abstract semantics that approximates the concrete semantics of the program and is efficiently computable. This abstract semantics can be chosen as the basis for a static analysis. Compared to an analysis of the concrete semantics, the analysis result may be less precise but the computation may be significantly faster.

A static analyzer is called *sound* if the computed results hold for any possible program execution. Abstract interpretation supports formal correctness proofs: it can be proved that an analysis will terminate and that it is sound, i.e., that it computes an over-approximation of the concrete semantics. Imprecision can occur, but it can be shown that they will always occur on the safe side. In runtime error analysis, soundness means that the analyzer never omits to signal an error that can appear in some execution environment. If no potential error is signaled, definitely no runtime error can occur: there are no false negatives. If a potential error is reported, the analyzer cannot exclude that there is a concrete program execution triggering the error. If there is no such execution, this is a false alarm (false positive). This imprecision is on the safe side: it can never happen that there is a runtime error which is not reported.

The difference between syntactical, unsound semantical and sound semantical analysis can be illustrated at the example of division by 0. In the expression $x/0$ the division by zero can be detected syntactically, but not in the expression a/b. When an unsound analyzer does not report a division by zero in a/b it might still happen in scenarios not taken into account by the analyzer. When a sound analyzer does not report a division by zero in a/b, this is a proof that b can never be 0.

B. *Astrée*

In the following we will concentrate on the sound static runtime error analyzer Astrée [13][34]. It reports program defects caused by unspecified and undefined behaviors according to the C norm (ISO/IEC 9899:1999 (E)), program defects caused by invalid concurrent behavior, violations of user-specified programming guidelines, and computes program properties relevant for functional safety. 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, violations of optional user-defined static assertions, violations of coding rules (MISRA C, ISO/IEC TS 17961, CERT, CWE) and code metric thresholds. In particular, this includes all error categories mentioned in Section II-C as principle security vulnerabilities.



Count	Name
806	Alarms
17	Invalid usage of pointers and arrays
2	Dereference of mis-aligned pointer
9	Possible overflow upon dereference
6	Out-of-bound array access
609	Invalid ranges and overflows
47	Float argument can be NaN or infinity
148	Overflow in conversion
324	Overflow in arithmetic
90	Overflow in conversion (with unpredic...
8	Invalid shift argument
8	Wrong range of second shift argument
57	Division or modulo by zero
42	Integer division by zero
15	Float division by zero
96	Invalid function calls
2	Reinterpreting incompatible paramete...
94	Stub invocation
10	Uninitialized variables
10	Use of uninitialized variables
2	Data and control flow alarms
2	Infinite loop
7	Invalid concurrent behavior
7	Read/write data race
6	Errors

Errors: 6
Alarms on code locations
Run-time errors: 776
Flow anomalies: 2
Rule violations: 0
Alarms on memory locations
Data races: 4
Reached code: 78%
Duration: 56min 53s

Figure 3. Astrée GUI with alarm overview

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).

The C99 standard does not fully specify data type sizes, endianness nor alignment, which can vary with different targets or compilers. Astrée is informed about these target ABI settings by a dedicated configuration file in XML format and takes the specified properties into account.

The design of the analyzer aims at reaching the zero false alarm objective, which was accomplished for the first time on large industrial applications at the end of November 2003. For keeping the initial number of false alarms low, a high analysis precision is mandatory. To achieve high precision Astrée provides a variety of predefined abstract domains, e.g., the interval domain approximates variable values by intervals, the octagon domain [35] covers relations of the form $x \pm y \leq c$ for variables x and y and constants c. The memory domain empowers Astrée to exactly analyze pointer arithmetic and union manipulations. It also supports a type-safe analysis of absolute memory addresses. With the filter domain digital filters can be precisely approximated. Floating-point computations are precisely modeled while keeping track of possible rounding errors.

To deal with concurrency defects, Astrée implements a sound low-level concurrent semantics [36], which provides a scalable sound abstraction covering all possible thread interleavings. The interleaving semantics enables Astrée, in addition to the classes of runtime errors found in sequential programs, to report data races, and lock/unlock problems, i.e., inconsistent synchronization. The set of shared variables does not need to be specified by the user: Astrée assumes that every global variable can be shared, and discovers which ones are effectively shared, and on which ones there is a data race. 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.

In some situations data races may be intended behavior. As an example a lock-free implementation where one process only writes to a variable and another process only reads from it may be correct, although there actually is a data race. However, a prerequisite is that all variable accesses involved are atomic. Astrée explicitly supports such lock-free implementations by providing means to specify the atomicity of basic data type accesses as a part of the target ABI specification. Data race alarms explicitly distinguish between atomic and non-atomic accesses.

Thread priorities are exploited to reduce the amount of spurious interleavings considered in the abstraction and to achieve a more precise analysis. A dedicated task priority domain supports dynamic priorities, e.g., according to the Priority Ceiling Protocol used in OSEK systems [37]. Astrée includes a built-in notion of mutual exclusion locks, on top of which actual synchronization mechanisms offered by operating systems can be modeled (such as POSIX mutexes or semaphores).

Programs to be analyzed are seldom run in isolation; they interact with an environment. In order to soundly report all runtime errors, Astrée must take the effect of the environment into account. In the simplest case the software runs directly on the hardware, in which case the environment is limited to a set of volatile variables, i.e., program variables that can be modified by the environment concurrently, and for which a range can be provided to Astrée by formal directives written manually, or generated by a dedicated wrapper generator. More often, the program is run on top of an operating system which it can access through function calls to a system library. When analyzing a program using a library, one possible solution is to include the source code of the library with the program. This is not always convenient (if the library is complex), nor possible, if the library source is not available, or not fully written in C, or ultimately relies on kernel services (e.g., for system libraries). An alternative is to provide a stub implementation, i.e., to write, for each library function, a specification of its possible effect on the program. Astrée provides stub libraries for the ARINC 653 standard, and the OSEK/AUTOSAR standards. In case of OSEK systems, Astrée parses the OIL (OSEK Implementation Language) configuration file and generates the corresponding C implementation automatically; in case of AUTOSAR projects the ARXML (AUTOSAR Extensible Markup Language) configuration file is used as an input.

Practical experience on avionics and automotive industry applications are given in [13][38]. They show that industry-sized programs of millions of lines of code can be analyzed in acceptable time with high precision for runtime errors and data races.

IV. CONTROL AND DATA FLOW ANALYSIS

Safety standards such as DO-178C and ISO-26262 require to perform control and data flow analysis as a part of software unit or integration testing and in order to verify the software architectural design. Investigating control and data flow is also subject of the Data Safety guidance [17], and it is a prerequisite for analyzing confidentiality and integrity properties as a part of a security case. Technically, any semantics-based static analysis is able to provide information about data and control flow, since this is the basis of the actual program analysis. However, data and control flow analysis has many aspects, and for some of them, tailored analysis mechanisms are needed.

Global data and control flow analysis gives a summary of variable accesses and function invocations throughout program execution. In its standard data and control flow reports Astrée computes the number of read/write accesses for every global or static variable and lists the location of each access along with the function from which the access is made and the thread in which the function is executed. The control flow is described by listing all callers and callees for every C function along with the threads in which they can be run. Indirect variable accesses via pointers as well as function pointer call targets are fully taken into account. Astrée also provides a call graph visualization enhanced by data flow information, which can be interactively explored (cf. Figure 4). Edges denote function calls, nodes in green represent functions free of alarms, nodes colored red correspond to functions with alarms, and nodes in yellow denote functions which call functions that cause run-time errors but do not cause potential run-time errors by themselves.

More sophisticated information can be provided by two dedicated analysis methods: program slicing and taint analysis. Program slicing [39] 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. Figure 5 shows Astrée's graphical visualization of a slice at the call graph level; green nodes represent functions which contain code which is part of the slice, gray nodes represent functions not contained in the slice.

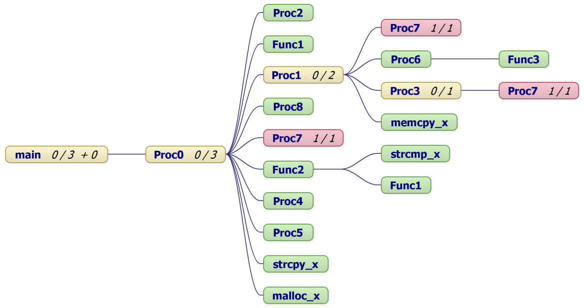


Figure 4. Astrée's Call Graph Visualization.

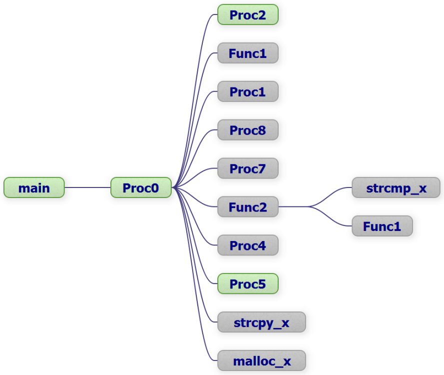


Figure 5. Astrée's Program Slice Visualization.

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. In the following we give a more detailed overview of both techniques.

A. Program Slicing

In this section we will first give a brief overview over classic static slicing. Then, we will introduce a new approach to slicing, which takes into account results from static program analysis via abstract interpretation, and describe its practical application based on an integration in Astrée. Finally, we describe the relevance of these concepts for demonstrating safety and security properties.

A slicing criterion of a program P is a tuple (s, V) where s is a statement and V is a set of variables in P. Intuitively, a slice is a subprogram of P which has the same behavior than P with respect to the slicing criterion (s, V). Computing a statement-minimal slice is an undecidable problem, but, using static analysis, approximative slices can be computed. A well-known algorithm for static slicing first computes a *system* *dependence graph* (SDG) and then solves a graph reachability problem on this graph [40]. The system dependence graph is an abstract description of the data- and control dependencies of a computer program.

The precision of this description directly influences the precision of slices computed from the SDG. Computing precise system dependency graphs is a non-trivial task since it requires deriving intricate program properties. These may include points-to information for variable and function pointers, code reachability, context information or possible variable values at certain program points. Astrée's core analysis computes invariants about such properties. We propose *analysis-enhanced slicing*, an approach which feeds some of these invariants to a program slicer contained in Astrée. This slicer can produce sound and compact slices by exploiting points-to and reachability information. A significant precision gain is achieved by reducing the amount of vertices and the amount of data- and control dependences in the system dependence graph. For simple static slicing, over-approximating the set of possible destinations of a pointer variable blows up the size of the system dependence graph as it may add false dependences to statements which contain variables that would otherwise not be included in the slice. This may cause a drastic transitive increase in the number of dependences and vertices. In contrast, interpreting invariants from Astrée's core analysis yields a precise local dependency description of a pointer dereference, which prevents the transitive blow up.

Astrée detects code which is guaranteed to be unreachable for any possible program execution. Ignoring such unreachable code fragments when constructing the system dependence graph further decreases its size. A system dependence graph computed by our approach is a sound abstraction of the data- and control dependences of a computer program. This follows from the soundness of the Astrée core analysis. As a consequence, the resulting slices are also sound. The amount of precision gain depends on the precision of the exported invariants.

We conducted experiments on programs from automotive and avionic industry in order to gauge the effectiveness of analysis-enhanced slicing. In the following, for each run of the analyzer and slicer we list the average execution time and memory consumption of three separate runs. Table I gives an overview of the programs under test. Exporting the invariants to be fed to the slicer does not significantly affect the performance of the analyzer. Run time and memory consumption increase by around 1% on average.

TABLE I. Projects

Project	#Lines	Reached Code	Analysis Time
Avionic1	417,723	98%	1h 41m 59s
Avionic2	71,566	73%	20m 38s
Automotive1	447,188	87%	52m 25s
Automotive2	1,623,140	17%	1h 12m 39s
Automotive3	10,331	92%	4s
Automotive4	1,705	83%	<1s

On these programs, the slicer has been executed in two different modes. In one mode it takes into account exported invariants (EXPORT) and in the other it does not (ALL). In the latter mode it assumes that each pointer may point to every variable of matching type. Table II and Table III show the execution time and memory consumption of the two slicer modes, respectively. For three programs the slicer requires more than 32GB in ALL mode and was stopped. In contrast, the slicer always terminates when considering exported invariants (mode EXPORT). In this mode the run time and memory consumption are much lower.

TABLE II. Slicing Efficiency - Run Time

Project	ALL	EXPORT
Avionic1	n/a	6s
Avionic2	23m 2s	1m 12s
Automotive1	n/a	1h 31m 20s
Automotive2	n/a	1h 13m 52s
Automotive3	2s	<1s
Automotive4	<1s	<1s

TABLE III. Slicing Efficiency - Memory

Project	ALL	EXPORT
Avionic1	>32,000 MB	691 MB
Avionic2	5015 MB	654 MB
Automotive1	>32,000 MB	5268 MB
Automotive2	>32,000 MB	1218 MB
Automotive3	87 MB	<1 MB
Automotive4	<1 MB	<1 MB

Finally, we show the number of lines of the computed slices in Table IV. In general, analysis-enhanced slicing yields significantly smaller slices.

TABLE IV. Slicing Precision

Project	ALL	EXPORT
Avionic1	n/a	661
Avionic3	70.817	67.134
Automotive1	n/a	48.380
Automotive2	n/a	39.507
Automotive3	4362	2868
Automotive5	415	176

So far the discussion of the Astrée slicer has been restricted to its context-insensitive mode. In this mode it always takes into account all contexts (call stack). While efficient, this constitutes another source of imprecision, since not all considered contexts describe actual execution paths. Therefore, the Astrée slicer supports a notion of context-sensitivity. Here, too, it relies on the precision of the Astrée core analysis. The call contexts computed during this analysis can be exploited in two different ways. Either all possible call contexts or a real subset of call contexts can be taken into account. Both modes exclude those function calls from the system dependence graph which do not match any of the specified contexts.

Considering all possible call contexts yields a sound slice with increased precision, when compared to the context-insensitive case. In contrast, the slice computed by taking into account a real subset of call contexts does not capture all possible behaviors of the original program which influence the slicing criterion. Instead, the behavior described by the slice is restricted to execution paths which match one of the specified call contexts. Depending on the choice of contexts, slices computed with this approach may be significantly smaller.

Analysis-enhanced slicing can be extended to context-sensitive slicing as well. Exploiting context-sensitive invariants in the same way as for context-insensitive slicing is sound and yields an increase of precision. In addition, it is possible to extend the framework by also exporting invariants separately per context. This is especially useful when considering only a small amount of contexts since in this case the invariants may be much more precise. Again, the resulting slice is possibly much smaller.

Another important advantage of analysis-enhanced slicing, in comparison with standard static slicing, is its efficiency. While computing sound slices with standard static slicing requires lots of time and memory, those resources are significantly lower for analysis-enhanced slicing. This is due to the smaller size and smaller complexity of the computed system dependence graphs. This efficiency improvement makes it possible to compute slices for very large programs in reasonable time.

In this work we do not consider dynamic slicing since a dynamic slice does not contain all statements potentially affecting the slicing criterion, but only those relevant for a specific subset of program executions, e.g., only those in which an error value can result. This restriction makes dynamic slicing unsuitable for proving program properties.

The different slicing modes presented in this section are relevant for demonstrating safety and security properties. Sound slices can be computed by context-sensitive analysis-enhanced slicing, when taking into account all possible contexts, or by context-insensitive analysis-enhanced slicing. With these slices it is possible to show that certain parts of the code or certain input variables might influence or cannot influence a program section of interest. They yield a global overview of these properties for the entire program.

In contrast to that, context-sensitive analysis-enhanced slicing, which only considers a subset of possible contexts, is more suitable for investigating the influence of a certain code section, e.g. a function, or a module, on the program location of the slicing criterion.

By considering exactly those contexts that pass through the interesting section, it is possible to prove that the program location of the slicing criterion may be influenced or cannot be influenced by this section. As the slices computed with this approach may be much smaller, this approach may yield much preciser results than investigation using sound slices.

B. Taint Analysis

In literature, taint analysis is often mentioned in combination with unsound static analyzers, since it allows to efficiently detect potential errors in the code, e.g., array-index-out-of-bounds accesses, or infeasible library function parameters [3], [22]. Inside a sound runtime error analyzer this is not needed since typically more powerful abstract domains can track all undefined or unspecified behaviors. Inside a sound analyzer, taint analysis is primarily a technique for analyzing security properties. Its advantage is that users can flexibly specify taints, taint sources, and taint sinks, so that application-specific data and control flow requirements can be modeled.

In order to be able to leverage this efficient family of analyses in sound analyzers, one must formally define the properties that may be checked using such techniques. Then it is possible to prove that a given implementation is sound with respect to that formal definition, leading to clean and well defined analysis results. Taint analysis consists in discovering data dependencies using the notion of taint propagation. Taint propagation can be formalized using a non-standard semantics of programs, where an imaginary taint is associated to some input values. Considering a standard semantics using a successor relation between program states, and considering that a program state is a map from memory locations (variables, program counter, etc.) to values in $\mathcal{V}$, the *tainted* semantics relates tainted states, which are maps from the same memory locations to $\mathcal{V} \times \{\text{taint}, \text{notaint}\}$, and such that if we project on $\mathcal{V}$ we get the same relation as with the standard semantics.

To define what happens to the *taint* part of the tainted value, one must define a *taint policy*. The taint policy specifies:

Taint sources which are a subset of input values or variables such that in any state, the values associated with that input values or variables are always tainted.

Taint propagation describes how the tainting gets propagated. Typical propagation is through assignment, but more complex propagation can take more control flow into account, and may not propagate the taint through all arithmetic or pointer operations.

Taint cleaning is an alternative to taint propagation, describing all the operations that do not propagate the taint. In this case, all assignments not containing the taint cleaning will propagate the taint.

Taint sinks is an optional set of memory locations. This has no semantical effect, except to specify conditions when an alarm should be emitted when verifying a program (an alarm must be emitted if a taint sink may become tainted for a given execution of the program).

A sound taint analyzer will compute an over-approximation of the memory locations that may be mapped to a tainted value during program execution. The soundness requirement ensures that no taint sink warning will be overlooked by the analyzer.

At first sight, it is easy to implement an efficient taint analysis: keeping track of the taint only requires one bit per variable. One bit means that iterating is fast, but when considering the whole set of variables, a sound analysis will require to compute fixpoints over bitvectors, meaning that in the worst case, convergence may take as many iterations as the number of variables in the program to analyze, each iteration requiring to compare two bit-vectors (again a linear cost). Fortunately, modern analyzers use advanced algorithmic techniques to efficiently compute such fixpoints, and basic taint analysis is way less complex than simple interval analysis.

A more precise taint analysis may be needed though: the typical case being taint cleaning functions that may fail to clean the tainting. In such cases, the function will usually return a value describing whether the cleaning succeeded. In order not to raise false alarms, it is then necessary to use a *relational* abstraction, keeping track of the relation between the taint bit, and the values of other variables. Doing so blindly leads directly to unscalable analyzes. In order to preserve the efficiency of the taint analysis, relations should only be kept between some taint values and some variable values, and only in a limited context. This leads to relational analysis through packing, a technique already in use for relational numerical domains, such as octagons or polyhedra, but which requires some expertise to fine-tune the heuristics. Finding the right packs depending on the parametric sources, cleaning and taint sinks is a research subject that will be our future work.

The tainted semantics can easily be extended to a mix of different hues of tainting, corresponding to an extension of the taint set associated with values. Then propagation can get more complex, with tainting not just being propagated but also changing hue depending on the instruction. This is needed not only to carry different taint analysis in one go, but also as a necessary semantic step to define some multi-level notions security breach. Carefully implemented, such extensions lead to a rather flexible and powerful data dependency analysis, while remaining scalable.

V. CONCLUSION

In this article, we have given an overview of 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 since the absence of such defects can be proven. In addition, 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] D. Kästner, L. Mauborgne, and C. Ferdinand, “Detecting Safety- and Security-Relevant Programming Defects by Sound Static Analysis,” in The Second International Conference on Cyber-Technologies and Cyber-Systems (CYBER 2017), ser. IARIA Conferences, J.-C. B. Rainer Falk, Steve Chan, Ed., vol. 2. IARIA XPS Press, 2017, pp. 26–31.

[2] MISRA (Motor Industry Software Reliability Association) Working Group, MISRA-C:2012 Guidelines for the use of the C language in critical systems, MISRA Limited, Mar. 2013.

[3] Software Engineering Institute SEI – CERT Division, SEI CERT C Coding Standard – Rules for Developing Safe, Reliable, and Secure Systems. Carnegie Mellon University, 2016.

[4] The MITRE Corporation, “CWE – Common Weakness Enumeration.” [Online]. Available: https://cwe.mitre.org [retrieved: Sep. 2017].

[5] Radio Technical Commission for Aeronautics, “RTCA DO-178C. Software Considerations in Airborne Systems and Equipment Certification,” 2011.

[6] IEC 61508, “Functional safety of electrical/electronic/programmable electronic safety-related systems,” 2010.

[7] ISO 26262, “Road vehicles – Functional safety,” 2011.

[8] CENELEC EN 50128, “Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems,” 2011.

[9] P. Cousot and R. Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in Proc. of POPL’77. ACM Press, 1977, pp. 238–252. [Online]. Available: http://www.di.ens.fr/~cousot/COUSOTpapers/POPL77.shtml [retrieved: Sep. 2017].

[10] D. Kästner, “Applying Abstract Interpretation to Demonstrate Functional Safety,” in Formal Methods Applied to Industrial Complex Systems, J.-L. Boulanger, Ed. London, UK: ISTE/Wiley, 2014.

[11] J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann, “Computing the worst case execution time of an avionics program by abstract interpretation,” in Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis, 2005, pp. 21–24.

[12] D. Delmas and J. Souyris, “ASTRÉE: from Research to Industry,” in Proc. 14th International Static Analysis Symposium (SAS2007), ser. LNCS, no. 4634, 2007, pp. 437–451.

[13] D. Kästner, A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, A. Schmidt, H. Hille, S. Wilhelm, and C. Ferdinand, “Finding All Potential Runtime Errors and Data Races in Automotive Software,” in SAE World Congress 2017. SAE International, 2017.

[14] D. Kästner, C. Rustemeier, U. Kiffmeier, D. Fleischer, S. Nenova, R. Heckmann, M. Schlickling, and C. Ferdinand, “Model-Driven Code Generation and Analysis,” in SAE World Congress 2014. SAE International, 2014.

[15] Wired.com, “The jeep hackers are back to prove car hacking can get much worse,” 2016. [Online]. Available: https://www.wired.com/2016/08/jeep-hackers-return-high-speed-steering-acceleration-hacks/ [retrieved: Sep. 2017]

[16] Y. Younan, W. Joosen, and F. Piessens, “Code injection in c and c++ : A survey of vulnerabilities and countermeasures,” Departement Computerwetenschappen, Katholieke Universiteit Leuven, Tech. Rep., 2004.

[17] SCSC Data Safety Initiative Working Group [DSIWG], “Data Safety (Version 2.0) [SCSC-127B],” Safety-Critical Systems Club, Tech. Rep., Jan. 2017.

[18] Radio Technical Commission for Aeronautics, “RTCA DO-178B. Software Considerations in Airborne Systems and Equipment Certification,” 1992.

[19] ——, “RTCA DO-333. Formal Methods Supplement to DO-178C and DO-278A,” 2011.

[20] ISO 26262, “Road vehicles – Functional safety – Part 6: Product development at the software level,” 2011.

[21] ISO/DIS 26262, “Road vehicles – Functional safety,” 2016.

[22] ISO/IEC, “Information Technology – Programming Languages, Their Environments and System Software Interfaces – Secure Coding Rules (ISO/IEC TS 17961),” Nov. 2013.

[23] MISRA (Motor Industry Software Reliability Association) Working Group, MISRA-C:2012 – Addendum 2. Coverage of MISRA C:2012 against ISO/IEC TS 17961:2013 ”C Secure”, MISRA Limited, Apr. 2016.

[24] MISRA (Motor Industry Software Reliability Association) Working Group , MISRA-C:2012 Amendment 1 – Additional security guidelines for MISRA C:2012, MISRA Limited, Apr. 2016.

[25] CERT – Software Engineering Institute, Carnegie Mellon University, “SEI CERT Coding Standards Website.” [Online]. Available: https://www.securecoding.cert.org [retrieved: Sep. 2017].

[26] MISRA (Motor Industry Software Reliability Association) Working Group, MISRA-C:2012 – Addendum 3. Coverage of MISRA C:2012 (including Amendment 1) against CERT C 2016 Edition, MISRA Limited, Jan. 2018.

[27] Wikipedia, “Blaster (computer worm).” [Online]. Available: https://en.wikipedia.org/wiki/Blaster_(computer_worm) [retrieved: Sep. 2017].

[28] CERT – Vulnerability Notes Database, “Vulnerability Note VU#720951 – OpenSSL TLS heartbeat extension read overflow discloses sensitive information.” [Online]. Available: http://www.kb.cert.org/vuls/id/720951 [retrieved: Sep. 2017].

[29] NIST – National Vulnerability Database, “CVE-2009-1888: SAMBA ACLs Uninitialized Memory Read.” [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2009-1888 [retrieved: Sep. 2017].

[30] J. Yang, A. Cui, J. Gallagher, S. Stolfo, and S. Sethumadhavan, “Concurrency attacks,” in In the Fourth USENIX Workshop on Hot Topics in Parallelism (HOTPAR12), 2012.

[31] M. R. Clarkson and F. B. Schneider, “Hyperproperties,” Journal of Computer Security, vol. 18, 2010, pp. 1157–1210.

[32] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” IEEE Journal on Selected Areas in Communications, vol. 21, no. 1, 2003, pp. 5–19.

[33] M. Assaf, D. A. Naumann, J. Signoles, E. Totel, and F. Tronel, “Hypercollecting semantics and its application to static analysis of information flow,” CoRR, vol. abs/1608.01654, 2016. [Online]. Available: http://arxiv.org/abs/1608.01654 [retrieved: Sep. 2017].

[34] A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, and C. Ferdinand, “Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée,” Embedded Real Time Software and Systems Congress ERTS^2, 2016.

[35] A. Miné, “The Octagon Abstract Domain,” Higher-Order and Symbolic Computation, vol. 19, no. 1, 2006, pp. 31–100.

[36] A. Miné, “Static analysis of run-time errors in embedded real-time parallel C programs,” Logical Methods in Computer Science (LMCS), vol. 8, no. 26, Mar. 2012, p. 63.

[37] OSEK/VDX, OSEK/VDX Operating System. Version 2.2.3., http://www.osek-vdx.org, 2005.

[38] A. Miné and D. Delmas, “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). IEEE CS Press, Oct. 2015, pp. 65–74.

[39] M. Weiser, “Program slicing,” in Proceedings of the 5th International Conference on Software Engineering, ser. ICSE ’81. IEEE Press, 1981, pp. 439–449.

[40] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural slicing using dependence graphs,” ACM Trans. Program. Lang. Syst., vol. 12, no. 1, Jan. 1990, pp. 26–60. [Online]. Available: http://doi.acm.org/10.1145/77606.77608