Safety-Critical Smart Systems with Software Coded Processing

Martin Süßkraut, SIListra Systems GmbH, Dresden, Germany
Jörg Kaienburg, SIListra Systems GmbH, Dresden, Germany

1 Introduction

Smart systems gain more and more importance in safety-critical applications. For instance, advanced driver assistance systems become more sophisticated and, hence, more complex by each generation. This complexity gain is driven by the evolution of smart systems. Modern lane keeping assistants exert an influence on the steering angles to keep cars on their current lanes. The emergency brake assistants activate the brakes even before a driver kicks the pedal. Other examples for smart systems in safety-critical applications are medical devices, e. g. infusion pumps used for intensive care, or autonomous robots in the automation industry.

When such smart systems malfunction they endanger passengers, road users and persons exposed to or in contact with the respective systems. Thus, any smart system built into a safety-critical application needs to be developed taking strictly into account the respective requirements of functional safety. These safety requirements are specified in industry standards like the IEC 61508 [2] and the ISO 26262 [5].

Inside a smart system of a safety-critical application, its so-called safety functions are the most sensitive software parts besides its functional program code that determines the function and behavior of the smart system. During development, the software development process embraces the development of the safety functions and additional safety measures, such as cyclic RAM checks and instruction set tests.

A common solution to safely execution the safety functions of a safety-critical system is to use hardware certified for safety-critical systems in combination with a structured safety-oriented software development process. The hardware of such systems typically consist of two main safety-critical hardware components: a CPU certified for safety-critical systems (incl. RAM and Flash) and a watchdog.

2 The Problem and the Solution

The problem we are focusing on is, that CPUs certified for safety-critical applications do not support all the requirements for modern smart systems. Typically, safety-critical CPUs do not provide all features that current non-safety critical CPUs provide. For instance, safety critical CPUs lack powerful GPUs, vectorization (SIMD) and modern efficient hardware architectures, e. g. pipe-lining. Hence, the more CPU and GPU resources a smart and safety-critical system requires, the more difficult it will be to implement such a system with safety-critical CPUs. Furthermore, the more powerful and featured a CPU is, the more difficult it becomes to develop instruction set tests, because these tests must cover the features of the used CPU.

The solution for the problem is to select a non-safety-critical CPU that fulfills the computational requirements of the smart system in combination with software techniques / procedures that provide the required safety features. As a result, a new kind of product is created with satisfies the performance *and* the safety requirements – allowing to potentially go beyond what can be realized today with common safety-related hardware components.

To achieve this, developers need to use appropriate software solution like “Diversified Encoding with Software Coded Processing” [1] to safely execute the safety functions. With the help of the diversified encoding, the watchdog can detect whether the CPU executes correctly all given safety functions. This solution solves the problem, because it does not require a CPU that is certified for safety-critical systems.

In this paper, we discuss how to build a safe smart system with a Raspberry Pi, diversified encoding and a watchdog. The Raspberry Pi is a powerful embedded system with a Broadcom BCM2835 processor based on ARMv11 and a VideoCore GPU. Both the Raspberry Pi and the ARMv11 processor are *not* hardware devices which have been specifically designed for safety-critical purpose. The watchdog uses the challenge-response approach and does not differ from solutions that are used in existing safety-critical smart systems.

3 Diversified Encoding with Software Coded Processing

3.1 Software Coded Processing

Software Coded Processing (SCP) is the primary part of diversified encoding. Once SCP becomes part of a smart system, this software technology detects execution errors inside this smart system. As a result, any smart system that is properly equipped with SCP becomes safer and more reliable.

To integrate SCP into a smart system, its software program has to be modified. This modification can be either done manually or by using an automated code transformation tool. The result of both ways of program modification is a new version of the former program code. This new version is functionally identical to its original program code and, in addition, is capable to detect the occurrence of execution errors – continuously during the system's run-time and independent of its explicit hardware.

To achieve the detection SCP *encodes* the complete data flow of a program, i. e. all constants, variables and operations. Encoding means SCP changes the representation of the data to additionally include redundancy. The literature describes different ways of encoding. The most prominent encoding is the AN encoding: Every value in the program is multiplied by a constant A [3]. Values that are not multiples of A are considered as invalid. Once SCP has been applied to a program code, all operations in such a modified program must work with encoded values. We call the modified program the *encoded program*. The original unmodified program is the *native program*.

Example: A given program calculates 2 + 3. Using A = 7, the encoded program calculates 14 + 21. Without any execution error, the result is a valid value: 35, which is a multiple of 7. Two kinds of execution errors can influence the result of this example:

- One of the input values is an invalid value. This can either happen because of a bit-flip in the memory, that stores the value, or when the value was already the result of an erroneous computation. For example, if the 14 changed to a 13, the result will be 34 which is not a multiple of 7 and, hence, an invalid result.
- The operation itself can be incorrect. For example, the addition could add an additional 2 to the result. The result would be 37, which is not a multiple of 7 and, therefore, not a valid result.

An encoded program could check any encoded variable at any point in time. However, for performance reasons it is better to only check the output values (Section 5). Checks in-between are not necessary because an operation using an invalid value produces an invalid result. Hence, the error propagates through the data flow of the program.

The main differences between the native program and the encoded program are:

- **Data types**: The data types change from 32-bit to 64-bit. SCP needs the additional bits for information redundancy. For instance, the largest 32-bit unsigned value is 4,294,967,295. Once encoded with A = 7, this value turns into 30,064,771,065. This value does not fit into 32-bit.
- **Constants**: All constants must be AN encoded.
- **Operations**: Encoded operations replace all native operations. Encoded operations operate on encoded values like native operations operate on native values.

The safety of the AN encoding depends on the constant A. The following properties of A influence the safety of AN encoding:

- **Size:** The larger A the more safe is the encoding. To illustrate, we assume, that n is the number of bits one requires to represent all native values (e. g. 32-bit) and k is the number of bits to represent A (e. g. 3 bits for A = 7). To represent all encoded values one needs n + k bits. However, the number of valid encoded values is the same as the number of all native values: 2^n. 2^{n+k} values are representable with n + k bits. However, only 2^n of all words representable with n + k bits are valid words. All other words are invalid. If an execution error changes a valid value into a different valid value, this error is not detected. The probability for a completely random error to change a valid value into another valid value is approximately 2^n / 2^{n+k} = 1 / 2^k. The probability 1 / 2^k depends only on the size of A.
- **Hamming Distance:** Because computers represent values as binary words, the hamming distance of the AN encoding is also important. The hamming distance depends on the value of A. For example, A should never be a power of 2, because the resulting AN encoding has the hamming distance of 0. The reason is that multiplying with a power of two is the same as a shift. In fact, A should be odd since the prime factor of two within A does not contribute to the hamming distance at all.

3.2 Diversified Encoding

Diversified encoding is one way to apply SCP. It is based on two divers executions of the same safety function and, because of its specific construction, it supports industry standards like the IEC 61508 [2] (Figure 1).

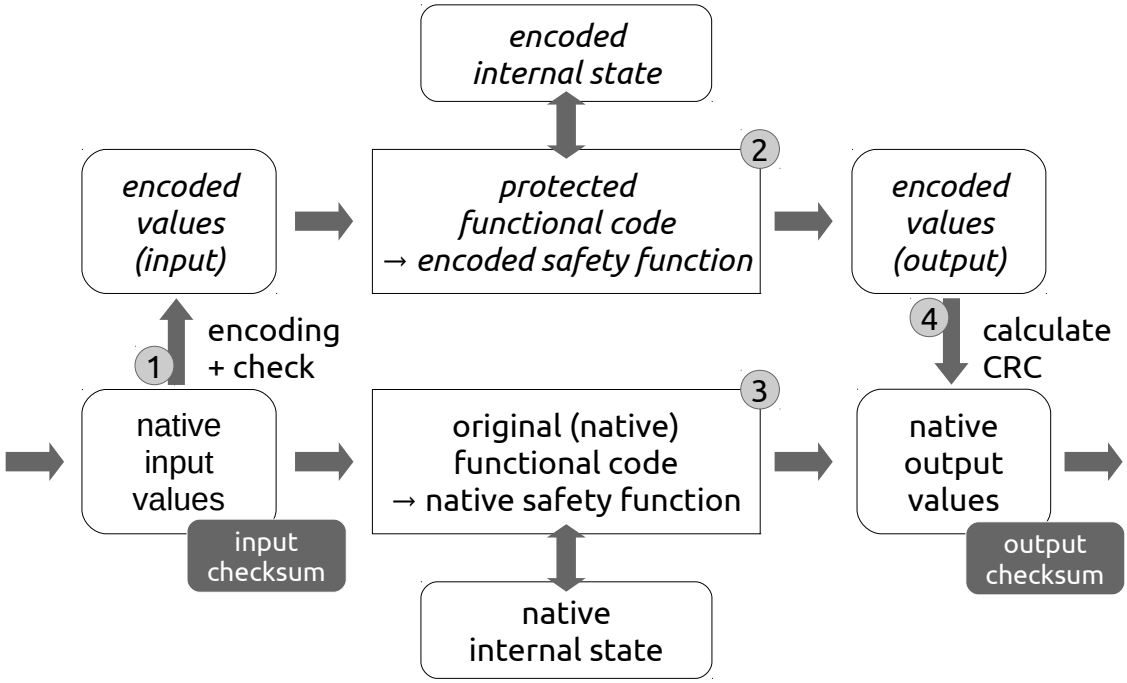

Fig. 1: Diversified encoding – principle of operation.

The two diversified executions are:

- **Native Execution:** The native execution is the computation of the original source code of the safety function. This source code operates on the native input values and the native states. The native execution only changes the native state. The results of the native execution are the native outputs.
- **Encoded Execution:** The encoded execution is on the computation of the encoded variant of the safety function. This requires that the original safety function source code has to be transformed and encoded upfront. Then, the encoded execution operates on encoded input values, on the encoded state, and it produces encoded outputs.

Both executions operate on the same values. The encoded input values are the encoded variants of the native input values. The same applies to the state and output values.

The aforementioned two executions are managed by a diversity framework. This framework is generated from the source code of the original safety functions plus the required annotations.

The diversity framework generates two checksums: over the native output values and over the encoded output values. After executing the safety function, the checksums

over the native and encoded output values get compared. Identical checksums indicate that the native output values are correctly computed, i. e. they are generated by an error-free execution. Deviating checksums are interpreted as an execution error during the respective computation.

Fig. 1 shows the principle of operation and, during operation, the data flow of one execution of a safety function. The data flow starts at (1) with the native input values. As soon as the native input values are assembled, these input values get protected by calculating a checksum for them: the input checksum. The diversity framework encodes and checks the native input values. The result is a set of encoded input values. The diversity framework uses the input checksum over the native input values to check the correctness of the encoded input values.

Next, in step (2), the diversity framework executes the encoded safety function. The encoded safety function reads the encoded input values and the encoded internal state. It performs its calculations, updates the encoded internal state, and produces the encoded output values. In step (3), the diversity framework executes the native safety function. The native safety function reads the native input values and the native internal state. It updates the native internal state and produces the native output values. In the last step (4), the diversity framework calculates the checksum of the native output values over the encoded output values. Then, the checksums of the native and encoded output values are compared.

So far we explained the detection of execution errors within the data flow of the safety function, i. e. within the safety function's calculations. Additionally, execution errors in the decisions of the safety function, i. e. the safety function's control-flow need to be detected. The control flow of the encoded safety function and the native safety function are similar. In order to safely detect control flow errors, control flow checks can be implemented into the encoded safety function. State-of-the-art control flow checks [4] support any C99 control flow including function calls, if-else, for-loops, while-loops, do-while-loops, break, continue, switch and even goto. The diversity framework integrates the data flow of the control flow checks in a way, that every control flow error changes the output checksum.

4 Example: A Safety Critical Application With a Raspberry Pi

The safety critical application running on the Raspberry Pi in our example application grabs a video stream frame by frame from a camera, enhances it with image processing, and sends it out over Ethernet, e. g. for displaying. Fig. 2 shows the architecture of the example application.

One typical safety function is the detection of repeated or missing frames. A human might not recognize early enough repeated frames or a frozen video stream. Therefore, the safety check must be taken over by the smart system. For instance, it could use a time stamp from the camera to detect repeated frames like a message counter in message passing systems (e. g. on a CAN-bus). Any other hardware failure, such as corrupted frames and errors in the image processing, lead to a corrupted video output which could be recognized easily by the spectator. Hence, safety functions could be omitted for such kind of calculations.

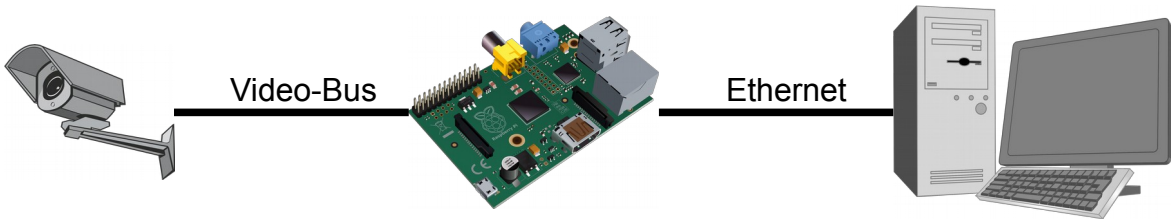

Fig. 2: Architecture of example application.

In this example, a safety function “time stamp check” shall be assumed as the detector for repeated or frozen frames. The cyclic RAM checks and CPU self-tests can be omitted when using SCP, because SCP detects memory corruption and malfunctioning instructions as execution errors. Hence, the development can focus on the safety function, but does not need to develop cyclic RAM checks and instruction set tests. The source code below shows the interface of the exemplary time stamp check. It gets a pointer to the current frame as input and returns a value indicating whether the time stamp in the frame is correct or not.

```
enum CheckResult {
  ok = 0xAAAAAAAA,
  failure = 0x55555555
};

enum CheckResult TimeStampCheckImpl(Frame* frame);
```

The time stamp check returns `ok`, when the time stamp in the frame is valid, i. e. no frame is repeated or skipped. Otherwise, the time stamp check returns `failure`.

Next, the transformed source is generated to implement the hardware execution error detection. As aforementioned, this transformed code provides the same functionality as the original code and, in addition, provides a checksum over the return value. The time stamp check gets a new interface because now it additionally needs to return the output checksum:

```
enum CheckResult TimeStampCheckSafe(uint32_t* checksum, Frame* frame);
```

An execution error that changes the value returned from `TimeStampCheckSafe` will be detected by comparing the output checksum with the checksum over the return value. The diversity framework contains the function `SIListra_diversity_output_checksum(returnValue)` that calculates the checksum over the return value.

5 Integration of a Challenge-Response-Watchdog

Challenge-Response-Watchdogs (CRWD) are a feasible method to monitor safety systems. A CRWD sends challenges to the safety system and expects a correct response within a defined time frame. If the response is wrong or does not arrive in a timely manner, the CRWD initiates the appropriate safety reaction for the system, e. g. stop, reset, or fail-over.

Challenges and responses are usually integer values. For each possible challenge, the expected response has to be pre-calculated. The expected responses are than stored within the watchdog as part of the watchdog's configuration.

The following source code example shows how a response to a given CRWD challenge is generated (here: unsigned 32-bit integer variables). Whenever the CRWD sends a new challenge it writes the variable `challenge`. Additionally, the CRWD clears the `response` variable. In the moment the CRWD expects a response, it reads the variable `response` and compares its value to the expected response.

```
extern uint32_t challenge;
extern uint32_t response;

void TimeStampCheck(Frame* frame) {
  uint32_t checksum = 0;
  enum CheckResult res = TimeStampCheckSafe(&checksum, &frame);
  response = challenge + checksum +
             res - SIListra_diversity_output_checksum(res);
}
```

When no execution error influenced the output value `res`, the values of `checksum` and `SIListra_diversity_output_checksum(res)` are the same. Hence, the expected response for a given challenge is `challange + ok`. If the time stamp check fails without an execution error it returns `failure`. In this case the wrong response is calculated and the watchdog responds with the safety reaction as expected.

When the values of `checksum` and `SIListra_diversity_output_checksum(res)` differ because of an execution error, the value of `response` deviates from the expected response value. Also when the safety function is not executed at least once between two subsequent challenges and the expected time for a response, the `response` variable contains an invalid response value. Hence, the CRWD detects situations when the safety functions is not correctly executed and when an execution error influences the execution of the safety function.

The presented CRWD solution always overwrites the `response` variable. Thus, an error-free execution of the safety function might mask a previous error if the watchdog did not check the `response` between these two executions. However, one can avoid the this problem by propagating response values between two executions.

6 Conclusion

This paper illustrates how a safety-critical application can be realized with a non-safety-critical device (a Raspberry Pi) in such way that it is capable to fulfill safety requirements. SCP provides the technology to detect hardware execution errors on the Raspberry Pi.

SCP in combination with diversified encoding is a flexible and hardware-independent solution to detect execution errors with the potential to fulfill ASIL-D requirements. ASIL-D is the highest safety level of the ISO 26262 [5]. One can also achieve SIL3 and SIL4 of the IEC 61508 with SCP. For SIL4 the IEC 61508 additionally requires hardware redundancy.

This examples illustrates three key points:

- It is possible to use non-safety-critical hardware for safety critical applications with the help of SCP.
- To integrate diversified encoding one needs just a generic watchdog. The watchdog integration is also generic and does not depend on specific safety function.
- SCP can replace many currently needed measures like cyclic RAM checks and instruction set tests.

Thus, the software developer can focus on the original safety function. That makes SCP a flexible and generic solution for developing future smart systems, especially for safety-critical applications.

In conclusion, diversified encoding with SCP solves the problem of the unavailability of feature-rich and powerful CPUs that are certified for use in safety-critical systems. Diversified encoding with SCP enables the use of modern CPUs that are not certified for safety-critical systems such as many ARM CPUs. The possibility to use such modern CPUs in smart systems enables the development of a new generation of smart systems that can be smart, powerful, and safe at the same time.

7 References

[1] Safe Program Execution with Diversified Encoding; Martin Süßkraut, André Schmitt, Jörg Kaienburg, Embedded World Conference 2015.

[2] IEC 61508:2010 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems (E/E/PE).

[3] AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware (Ute Schiffel, Martin Süßkraut, Christof Fetzer), In SAFECOMP '09: Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security, Springer-Verlag, 2009.

[4] ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software (Ute Schiffel, André Schmitt, Martin Süßkraut, Christof Fetzer), In Computer Safety, Reliability, and Security (Erwin Schoitsch, ed.), Springer Berlin / Heidelberg, volume 6351, 2010.

[5] ISO 26262 Road vehicles – Functional safety. 2011.