Safe Program Execution with Diversified Encoding

Martin Süßkraut, André Schmitt, Jörg Kaienburg
SIListra Systems GmbH
Dresden, Germany
{martin.suesskraut, andre.schmitt, joerg.kaienburg}@silistra-systems.com

***Abstract*—Currently, hardware designed and certified for safety-critical systems is one important building block for any safety-critical application. Such hardware provides the detection of execution errors. However, many modern safety-critical applications, like autonomous driving, require features and performance levels that are not available from safety-certified hardware. The solution to this problem is to use hardware that is not certified for safety-critical systems, for example consumer-graded hardware, but fulfills the requirements. Additionally, a software solution provides the detection of execution errors.**

This paper introduces such a software solution called "Diversified Encoding with Coded Processing". Due to its hardware-independence, this solution provides the flexibility to build safety-critical systems from non-safety-critical hardware components. This solution can be automated with a code transformation tool to further increase the flexibility.

Keywords—safety, ISO 26262, IEC 61508, software coded processing, software diversification

I. Introduction

Traditionally, safety-critical hardware is designed for detecting execution errors. Current non-safety-critical hardware or consumer-graded hardware, respectively, are typically unable to detect execution errors but provide significantly higher performance. Hence, safety-critical systems that require high performance or special features that are only available in consumer-graded hardware, e.g. GPUs, need a new solution for detecting execution errors.

This paper gives an introduction into the software solution called "Diversified Encoding with Software Coded Processing": A solution to continuously detect execution errors at runtime and allowing to make any system intrinsically safe. Because, diversified encoding is hardware-independent it can be used with consumer-graded hardware. After the introduction in Section I, Section II introduces an examples which will be used during the complete paper. Sections III and IV introduce software coded processing and diversified encoding, respectively. Section IV also contains the safety arguments for diversified encoding.

ISO 26262 lists coded processing in Volume 5 under D 2.3.6 as a technology to detect hardware errors [5]. IEC 61508 lists coded processing in volume 2 in A.4 [4]. Coded processing in combination with diversified encoding is a flexible solution to detect execution error with the potential to fulfill ASIL-D and SIL4 requirements. ASIL-D and SIL4 are the highest safety levels of the ISO 26262 and IEC 61508, respectively. For SIL4, the IEC 61508 additionally requires hardware redundancy. The utilization of diversified encoding via a fully automated code transformation tool allows the efficient deployment of diversified encoding across different industries, e.g. automotive, medical, automation, rail, and avionics.

Section V introduces a code transformation tool for generating the diversified encoding solution. To achieve this, the tool takes the source code of such a safety function as its input. The output of the tools is again source code. This generated source code provides the same functionality as the given safety function plus the capability to detect hardware errors.

The novelty of the presented tool is the broad support for the C programming language. The diversified encoding solution automatically safe-guards against transformation failures of any transformation tool. Previous approaches using coded processing do not have these properties.

The paper concludes in Section VI with a discussion of the advantages of the diversified encoding approach: hardware-independence, flexibility of software and tool support for the C programming language. As one of the intended side effects, diversified encoding decouples hardware-error detection and functionality. Major advantages of diversified encoding are hardware-independence and higher flexibility in setting up system architectures. Hence, development and innovation cycles could be expedited. The presented technique of diversified encoding with coded processing allows system architectures equipped with safety functions while running on non-certified hardware – something that yet has not been possible.

II. EXAMPLE

The following exemplary safety function will help to explain and demonstrate the diversified encoding and its properties and advantages. This example focuses on the overall approach of diversified encoding and code generation.

The example safety function is a safe counter. It has the following specification:

1. The safe counter is one shared 32-bit counter unsigned integer value.
2. The initial counter value is 1.
3. The safe counter has an interface to query the current counter value and to increment the counter value by a given unsigned integer value.
4. Execution errors while executing the safe counter must be detected.

For a real world application, one needs to extend this specification to include aspects like integer overflow and concurrency.

The software developer designs an interface from this specification: `uint32_t incSafeCounter(uint32_t incValue)`.

The implementation itself does not contain any specific measures to detect execution errors. Sections III and IV demonstrate how to automatically generate the source code to detect execution errors. Especially Section IV.C shows how to change the interface to enable the detection and handling of execution errors. The following source code shows a possible implementation of this safety function (with C99 fixed-width integer types[1]):

```
/* include standard C library
   for fixed-sized integer types */
#include <stdint.h>

/* the safety critical counter variable */
static uint32_t safeCounter = 1;

/* interface of the counter */
uint32_t incSafeCounter(uint32_t incValue) {
  safeCounter += incValue;
  return safeCounter;
}
```

III. SOFTWARE CODED PROCESSING

Diversified encoding builds on the coded processing technology [1]. Software coded processing (SCP) adds information redundancy to a software program to enable it to detect execution errors. To integrate SCP into a software program one has to either rewrite the program manually or one can use an automated transformation tool. SCP is applied to the complete data flow of a program. In other words, all constants, variables and operations have to be re-programmed while making use of SCP.

[1] The type `uint32_t` is a 32-bit unsigned integer. The type `uint64_t` is a 64-bit unsigned integer.

The literature describes different encodings. The most prominent encoding is the AN encoding: Every value in the program is a multiple of a constant A [2]. AN is the name of the encoding and not an abbreviation. Values that are not multiples of A are considered as invalid. With SCP, all operations in a program must work with these encoded values. An execution error produces invalid values.

A helpful example is the summation 2 + 3. Encoding with A = 7, the calculation turns into 14 + 21. Without an execution error, the result is 35 and valid value because it is a multiple of 7. The criterion which decides whether the result is valid or invalid is the property that valid results are a multiple of A. If not, results are considered as invalid.

Two principle kinds of execution errors can influence the result:

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

An encoded program can check any encoded variable at any point in time. However, for performance reasons it is better to only check the output values (see Section IV.E). In-between checks are not necessary because the error propagates through the data flow of the program.

The source code below shows the example from Section II with a simplified AN encoding for A = 7:

```
/* include standard C library
   for fixed-sized integer types */
#include <stdint.h>

/* 1 is AN encoded 7 */
static uint64_t safeCounter = 7;

uint64_t incSafeCounter_encoded(
        uint64_t incValue)
{
  safeCounter =
      add_encoded(safeCounter, incValue);
  return safeCounter;
}
```

The main differences between the native safety function and the encoded safety function are:

- **Data types:** The data types change from `uint32_t` to `uint64_t`. SCP needs additional bits for the information redundancy. For instance, the largest 32-bit unsigned value 4,294,967,295 encoded with A = 7 is 30,064,771,065. This value does not fit into 32-bit.

- **Constants:** All constants must be AN encoded. In the example, the initialization value of `safeCounter` is now 7 – the AN encoded 1.
- **Operations:** Encoded operations replace all native operations. Encoded operations operate on encoded values like native operations operate on native values. For instance, the `add_encoded` returns the encoded sum of its two encoded input 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. If n is the number of bits required 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) n + k bits are needed to represent all encoded values. The number of valid encoded values is the same as the number of all native values: 2^n. Only these 2^n out of all words representable with n + k bits are valid words. All other words are invalid. The probability for a completely random error to change a valid word into another valid word is roughly $2^n/2^{n+k} = 1/2^k$. 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.

AN encoding is not the only encoding for SCP. Examples for other encodings are the ANB and the ANBD encoding [3]. Both are built on AN and introduce encoding parameters B and D additional to the encoding parameter A. The goal of these encodings is to detect execution errors that cannot be detected with AN alone. For example, if – due to an execution error – an addition changes into a subtraction, the result is still a valid AN encoded value.

The additional encoding parameters increase the complexity of the encoding and, hence, they increase the required CPU resources. The solution to detect execution errors is the diversified encoding (Section IV). One of the advantages of the diversification is that its complexity and required CPU resources are lower than with ANB and ANBD encodings.

IV. Diversified Encoding

A. Overview

Diversified encoding is based on two distinct executions of the same safety function. These two executions are:

- **Native Execution:** The native execution is the result of the original source code of the safety function. This source code operates on the native input values and the native state. The native execution only changes native state. The result of the native execution is the native output.

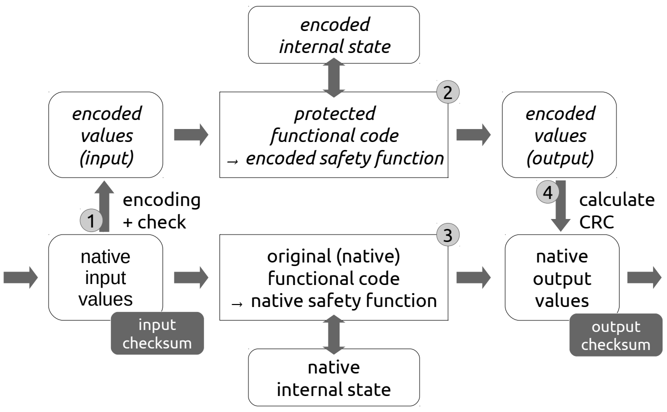


Fig. 1. Data-flow model of diversified encoding.

- **Encoded Execution:** The encoded execution is based on the encoded variant of the safety function. The encoded execution operates on encoded input values and on the encoded state. It produces an encoded output.

Both executions are completely distinct computations but operate on the same values. The encoded input values are the encoded variants of the native input values. The source code of the original, native code is used to generate the encoded source code thereof. This can be done either manually or, recommended due to the high degree of reproducibility, via a code transformation tool.

A diversity framework manages both executions. The transformation tool generates the source code of the diversity framework from the source code of the original safety functions.

The component that uses the safety function can detect and handle execution errors with the help of checksums. We call such a component a caller. The diversity framework generates two checksums: one over the native output values and another over the encoded output values. A caller operates solely on the native input and output values. After executing the safety function, a caller must compare the checksum over the native and encoded output values to verify whether the native output values are from an error-free execution. Section IV.E shows an example for such a check. If these checksums differ, an execution error has happened.

Fig. 1 shows the dataflow of a safety function with the diversified encoding solution. The data flow starts at (1) with the native input state. The native input state is protected by a checksum. A caller must calculate this checksum, as soon as the caller has assembled the input state. Section IV.D contains a detailed discussion of the native input values and the checksum. When a caller executes the safety function, it passes the control to the diversity framework. In step (1), the diversity framework encodes and checks the native input values. Step (1) produces the encoded input values for step (2). 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 diversity framework passes the control back to the caller. It's the caller's responsibility to check the checksum of the native output values.

The safety function works with the following parts:

- native input values
- input checksum
- native safety function
- diversity framework (including the encoded safety function) generated by the code transformation tool
- native output values
- output checksum

The code generation tool generates the source code of the steps (1), (2) and (4). Steps (1) and (4) are part of the diversity framework. Step (2) is the encoded safety function. The source code of step (3) is the original source code of the native safety function.

The control flow of the native and encoded safety functions are similar. In order to safely detect control flow errors, the code transformation tool can generate control flow checks into the encoded safety function. These control flow checks build a separate data flow which mirrors the control flow. A control flow error influences only the control flow and not this separate data flow. A caller detects control flow errors, because the result of the data flow does not match the control flow. The diversity framework integrates the data flow of the control flow checks in a way, that every control flow error invalidates the output checksum. State-of-the-art control flow checks support any C99 control flow including function calls, if-else, for-loops, while-loops, do-while-loops, break, continue, switch and even goto.

B. *Safety Argument for Single Execution Errors*

Execution errors can either happen transient, e.g. as soft error due to radiation, or permanent, e.g. due to hardware aging. Transient errors influence by definition only a small part of an execution. Permanent errors are also called common cause errors. They influence larger parts or several parts of an execution in the same way. Execution errors can influence three aspects of an execution: the data flow, the control flow, and the timing. The data flow is the data stored in a system together with all calculations (arithmetic, comparison, etc.) that a safety function performs on this data. The control flow are all decisions that a safety function contains (loops, function calls, etc.). The timing is the timely execution of the safety function. Execution errors can interfere with the timing by making the execution too slow or stopping the execution completely. In this sense, a crash is also a timing error.

Execution errors without influence on the result of a computation, i.e. the output values of the computation, can be ignored.

The safety argument starts with transient execution errors. Fig. 1 shows the four steps of the data flows within the diversified encoding. All four steps happen in sequence. Hence, any transient error – according to the definition of transient – influences exactly one of these steps. The following discussion argues for each step individually, that a caller detects any transient error happing in this step when the caller checks the output checksum.

- Step (1) encodes the input values. Any transient error can either falsify the native input value or the encoded input value. In case the native input value is erroneously changed, the input checksum does not match the native input values any more. The same is true, when the transient error changes encoded input values. The diversity framework propagates this mismatch to the output checksum. Therefore, the diversity framework first adds the encoded checksum of the encoded input state to the output checksum and then encodes and subtracts the input checksum. Without any execution error, the output checksum is not changed and both checksum values are identical. And in case the execution error produces invalid encoded input values, the diversity framework detects this error because the encoded input checksum becomes invalid.
- Any error that influences the execution of the encoded safety function propagates to the encoded output value and/or to the encoded state. If the error influences none of them, there is no effect on the execution of the encoded safety function. Then, the error can be ignored. When the error influences the encoded output, the diversity framework will generate an output checksum that does not match with the checksum over the native output values because the latter ones were not influenced by the transient error. But in case the error influences the encoded state only, the error remains dormant. The output checksum will be correct because the encoded output values are correct. The error can propagate (depending on the implementation of the safety function) to the output in a later execution. But it will be detected then. It is important to note the fact that the dormant error in the encoded state can only propagate to the encoded output and nowhere else. When it propagates it will be detected because it falsifies the output checksum. As long as the error does not propagate into the encoded output values it has no effect and, thus, can be ignored.
- The argument for any error that influences the execution of the native safety function follows the arguments of step (2). But now, the error influences the output values directly. Because the error is transient, it does not influence the encoded output values and, hence, it does not influence the output checksum. Therefore, the output checksum will not match the output values and the error will be detected. Again, the

error may only influence the native state and remain dormant. In this case, the argument for step (2) applies: In the first execution, where the dormant native state error influences the output values, the error will be detected.

- Step 4 only works on the encoded output values and produces the output checksum. Hence, any transient error here can only influence the output checksum. Therefore, the diversified encoding detects this transient execution error when the comparison of the output values with the (erroneous) output checksum is done.

In the moment a control flow error takes place within any part of the diversified encoding that influences the data flow of this part, the data flow arguments above apply. A control flow error that does not influence the data flow has no effect and can be ignored. However, a control flow error can also skip or repeat steps of the diversified encoding. Steps (1) and (4) can be repeatedly executed without any negative effect. Repeated executions of steps (2) and (3) due to their internal states may either falsify their outputs and/or their internal states. Then, the aforementioned arguments about the data-flow errors apply. Skipping steps (1), (2) or (4) produces the wrong and perhaps old output checksum that does not match the output values. Skipping step (3) does not update the output values and, hence, the output values do not match the output checksum. A caller can detect the skipping of the whole execution of the safety function including the diversified encoding by setting the output checksum variable to a constant that is unlikely to be the next output checksum. In general, 0 is a good value for this constant.

A transient error can influence the timing, e.g. the safety function can execute too slow, stop, or crash because of a transient error. These cases can be detected with the help of a watch dog (see below).

In summary, the diversified encoding detects with a high likelihood all transient execution errors that influence the execution of a safety function with diversified encoding.

The second and last part of the safety argument discusses common mode execution errors. A common mode execution error influences more than one part of the execution in the same way. The steps (1), (2) and (4) of diversified encoding work on or produce encoded values. Step (2) works on native values. The calculation of the input checksum and the comparison of the output checksum work on native values. Because of the different representation of data between native and encoded parts, common mode execution errors are unlikely to falsify native and encoded part in the same way. Thus, common mode errors will produce with a high likelihood output values that do not match the output checksum. We explain this argument with two different examples:

- In the first example, it shall be assumed that the error consists of one or more bits of a register `r` which are permanently stuck-at 0. First, it is likely that the register `r` is used for different values in the native and encoded parts. In this case, both parts will be influenced differently. This is the case because a register `r` has a fixed bit width. Without loss of generality, it can be assumed that the bit width of `r` is 32. In the native part, a 32-bit value fits into `r`. In the encoded part, only half of the encoded value fits into `r`. Hence, it is very unlikely, that `r` falsifies both the native value and the encoded value in a way that the falsely encoded value matches the false native value. Furthermore, it is also unlikely that the stuck-at 0 bits in `r` produce a valid encoded value at all. For a deeper discussion about invalidly encoded values, Section III provides further information.

- In the second example, an execution error exchanges the addition operation against the subtraction operation for 32-bit values. For 32-bit values, the native part uses the 32-bit addition. The encoded part uses a 64-bit addition for the corresponding encoded values because a 32-bit native value becomes a 64-bit encoded value through encoding. Because of this fact, the faulty 32-bit addition is unlikely to produce matching native and encoded values in the native and encoded parts, respectively. Furthermore, the encoded parts use additional operations for control flow checks and for implementing encoded operations. Due to the nature of a common mode execution error, whenever a 32-bit addition is executed as part of the control flow checks or an encoded operation, this addition must turn into a subtraction. However, this will most likely either result in a detected control flow error or produce invalid values as a result of the encoded operation.

In summary, this safety argument concludes that diversified encoding detects both transient and permanent execution errors by comparing the output checksum with output values.

C. *Diversified Safe Counter*

This section explains the usage of diversified encoding with the example from Section II. The code transformation tool generates an entry point function `incSafeCounterSafe` point for the function `incSafeCounter`. This entry point is part of the diversity framework generated by the code transformation tool and it encapsulates the four tasks from Fig. 1:

1. It generates the encoded input values from the native input values.
2. It performs the native execution.
3. It performs the encoded execution.
4. It calculates the checksum over the encoded output.

The new entry point has the following interface: `uint32_t incSafeCounterSafe(uint32_t* checksum, uint32_t incValue)`. The new entry point has a new name, because the native entry point with the name `incSafeCounter` is still used in step (3). The return value and the parameter `incValue` have the same semantics as in `incSafeCounter`. A caller must provide a pointer to the checksum variable. The new entry point `incSafeCounterSafe` stores the checksum over the encoded output into this checksum variable.

To check for execution errors, e.g. when using the return value of `incSafeCounterSafe`, the caller's code must compare the value of the variable checksum with the checksum of the native output values. To calculate the checksum over the native output values, the transformation tool generates a `uint32_t SIListra_diversity_output_checksum(uint32_t returnValue)` function. The parameter `returnValue` must be the return value that `incSafeCounterSafe` returns. We exemplify the usage of the output checksum below with an integration of a watch dog.

To simplify the example, a checksum over the native input values was omitted. The following sub-section discusses an alternative to using native input checksums.

D. *Protection of Input Values*

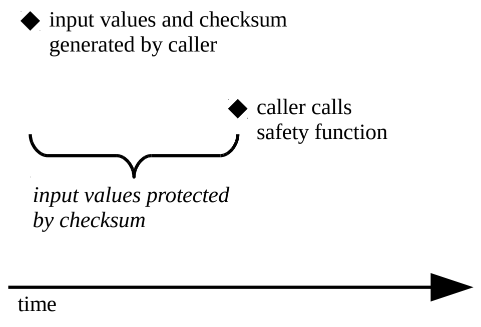
The input checksum solution can be insufficient depending on how the input values are generated. Fig. 2 (a) illustrates, that the checksum solution is a good fit, if all input value are generated by the caller at nearly the same point in time. Fig. 2 (b) shows a case were the input checksum solution is insufficient. When one or more other components provide the input values, the input values are unprotected, until the caller calculates the checksum. The caller can only calculate the input checksum when all input values are available. For example any bit-flip happening on an input value between its generation and the checksum calculation is not detected.

Fig. 2 (c) shows an alternative solution to determine the integrity of input values: Inverse storage. The component that generates an input value v stores this v into two variables. One variable contains v as it is. The other variable contains a bitwise negation of v. Both variables are part of the input values of the safety function. Within the safety function, an integrity check of both input variables has to be carried out. If the integrity is violated, then the safety function must take an appropriate action. Otherwise it continues to use only the original value.

The source code below shows the implementation of the example with inverse storage of the input value `incValue`. This is the source code of the original native safety function. The transformation tool generates the appropriate source code of the encoded safety function.

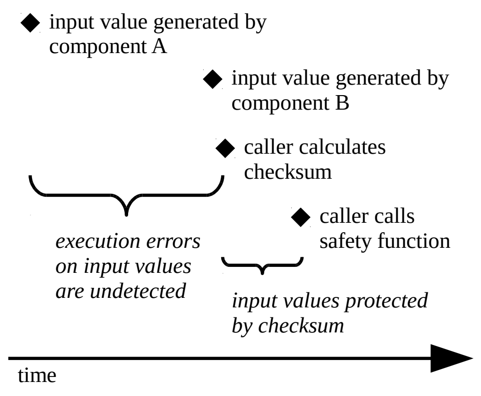
```
static uint32_t safeCounter = 1;

uint32_t incSafeCounter(
         uint32_t incValue,
         uint32_t incValue_invers)
{
  if (0xFFFFFFFF !=
         (incValue ^ incValue_inverse)) {
        /* for example trigger watch dog */
  }
  safeCounter += incValue;
  return safeCounter;
}
```

The parameter `incValue_inverse` stores the inverse input value of the parameter `incValue`. Using inverse storage changes the interface of the safety function as shown above.



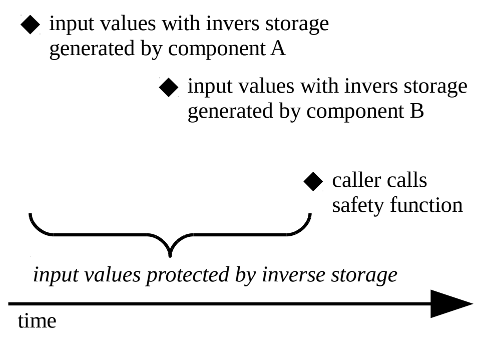
input values and checksum generated by caller
caller calls safety function
input values protected by checksum
time

(a) checksums protect input values from one component



input value generated by component A
input value generated by component B
caller calculates checksum
caller calls safety function
execution errors on input values are undetected
input values protected by checksum
time

(b) checksums protect input values from multiple components insufficiently



input values with invers storage generated by component A
input values with invers storage generated by component B
caller calls safety function
input values protected by inverse storage
time

(c) inverse storage protect input values from multiple component; checksum is not required

Fig. 2. Three variants for protecting input values. Protected means, that execution errors on the input values will be detected. Unprotected means, that there is a time window in which execution errors on some input values cannot be detected.

The transformation tool generates an entry point with the following interface: `uint32_t incSafeCounterSafe (uint32_t* checksum, uint32_t incValue, uint32_t incValue_inverse)`. Within the safety function, inverse storage is not necessary because the diversified encoding already protects the global variables (see global variable `safeCounter` in the example).

The example highlights two important properties of using inverse storage with diversified encoding:

- This solution protects the input values and the calculation from putting the input values into inverse storage throughout the execution of the safety function without a gap. Because the integrity check of the inverse stored input value happens within the safety function, execution errors within the check are detected by the diversified encoding technique. The checksum approach also protects the input values from checksum generation throughout the execution of the safety function. In some cases, the inverse storage solution is more flexible than the checksum solution because a component can put values into inverse storage without knowing anything about the input checksum of the safety function.

- Diversified encoding is a flexible solution that works together with traditional approaches to detect execution errors. A decision to use inverse storage for some or all input values is completely independent of the decision to use diversified encoding. Instead of inverse storage, other solutions can be used – such as replicating input values or having two separate input values and using a plausibility check within the safety function. Diversified encoding is compatible with these solutions.

E. Integration with a Watchdog

This section shows how to integrate diversified encoding with a challenge-response-watchdog (CRWD). A CRWD monitors a safety system. It regularly sends challenges to the safety system and expects a correct response within a defined time frame. In case the response is wrong or does not arrive in a timely manner, the CRWD initiates the appropriate safety reaction for the system, e.g. a stop, reset, or fail-over. Such CRWDs are state-of-the-art in the industry.

Challenges and responses are usually integer values. For each possible challenge, the expected response has to be pre-calculated. These expected responses are stored within the CRWD as part of the configuration of the system.

Our goal is to show how to enable the CRWD to monitor, that (1) the safety function is regularly executed and (2) that the executions are free of execution errors. The source code below shows how diversified encoding can generate a response to a given CRWD challenge. In the following discussion, any challenge and response shall be represented as unsigned 32-bit integer variables. The CRWD writes the variable `challenge` whenever it sends a new challenge. In addition, the CRWD clears the variable `response`, e.g. by setting it to an invalid response value. When the CRWD expects a response, it reads the variable `response` and compares its value to the expected response. In practice, the value of `response` can be merged with responses from other safety functions to calculate a unified response for the whole system.

```
/* constant for calculating the response for
the safe counter */
#define SAFE_COUNTER_SIG   (12u)
/* stores the current watchdog challenge */
extern uint32_t challenge;
/* stores the response for the watchdog */
extern uint32_t response;

/* initialize checksum variable with unlikely
checksum value (see above) */
uint32_t checksum = 0;

/* increment the safe counter by 1 with
inverse input (see above) */
uint32_t currentCounterValue =
    incSafeCounterSafe(&checksum, 1u, ~1u);
/* ... use currentCounterValue ... */

/* calculate response from challenge and
output checksum */
response = challenge + checksum +
    SAFE_COUNTER_SIG –
    SIListra_diversity_output_checksum(
        currentCounterValue);
/* ... */
```

When no execution error influenced the output value `currentCounterValue`, the values of `checksum` and `SIListra_diversity_output_checksum(currentCounterValue)` are the same. Hence, the expected response for a given challenge is `challenge + SAFE_COUNTER_SIG`.

When the values of checksum and `SIListra_diversity_output_checksum(currentCounterValue)` differ because of an execution error, the value of `response` deviates from the expected response value. Even when the safety function is not executed at least once between a new challenge and its corresponding time for a response, the variable `response` contains an invalid response value. Hence, the CRWD detects whether the safety functions is regularly executed and whether an execution error influences the execution of the safety function.

The presented solution always overwrites the variable `response`. Hence, a fault-free execution of the safety function might mask a previous error in case the CRWD did not check the `response` between these two executions. To avoid the masking of a previous error one can extend the solution to propagate an execution error through consecutive executions into the response that the CRWD reads.

V. Code Transformation

The aforementioned code transformation tool generates automatically the source code of the encoded program and the source code of the diversity framework. Of course, it is possible to write the encoded program and the diversity framework manually. But automatic code transformation provides the following advantages compared to manually implementing the encoded program and the diversity framework:

- **Development speed:** Changes to the original source code of the native safety function can be quickly adopted in the generated code. The alternative would be a time-consuming manual reprogramming.

- **Flexibility:** Requirements and parameters can be changed, elaborated, and examined very quickly. It only requires to rerun the code transformation tool. Without tool support, changes like enabling/disabling control-flow checks and changing the encoding would require a complete manual re-writing.

- **Correctness and reproducibility:** In most cases, a thoroughly tested code transformation tool works more correct than a human being. Diversified encoding additionally reduces the risk of tool errors. In case the code transformation tool generates source code for the encoded safety function that does not behave as the original source code, the output values of native and encoded safety function do not match. Hence, the diversified encoding per se detects any wrong output that results from an in-correct generated encoded safety function as execution error.

The code transformation tool works like a C compiler. It performs the following steps:

1. **Preprocessing:** The tool preprocesses the complete source code of the safety function including all included header files. Therefore, it must be configured with the same preprocessing `defines` as the C compiler.

2. **Parsing:** The tool generates an abstract syntax tree from the preprocessed source code including a semantic analysis. For instance the abstract syntax tree needs a complete type analysis. At this step, the tool can detect unsupported C languages features. If the tool does not support a language feature, it produces an error message and aborts the transformation process.

3. **Encoding:** Sufficient semantic information are available after the parsing step to encode the safety function. The tool replaces constants, variables, and operations with their encoded versions.

4. **Code Generation:** The result of the encoding step is an intermediate result. The last step is the final C code generation.

The current status of the tool allows to encode most C99 features. The following features are known to be supported by a state of the art code transformation tool:

- all integer arithmetic including logical operations and comparisons

- all control flow constructs of C99 from if-else, including loops, function calls to break and continue (except setjmp and longjmp)

- arrays, structs and any other pointer arithmetic

For each native C module that the tool gets as input, it produces an output C module containing the encoded version of the native C module. The generated C code must then be further processed in the tool chain, typically by the C compiler.

Optimizations in the C compiler cannot remove the encoding. The fact that all values are a multiple of an encoding parameter A is not visible to the C compiler because the C compiler processes each C module individually. Even with link time optimization, which provides the optimizer the view on all C modules at once, the optimizations cannot remove the encoding. The optimizations would have to prove that the whole encoded safety function is equivalent to the native safety function, including all checksum calculations in the diversity framework. No state-of-the-art compiler provides such optimizations. In addition, fault injection could be used to demonstrate that the encoding has not been removed during compilation.

Besides encoding itself, the code transformation tool can also generate the diversity framework. The diversity framework consists of the source code of the functions that calculate the native input checksum and the native output checksum and the source code of the new entry points.

VI. Discussion and Conclusion

A. *Scope of Detection*

Diversified encoding is a probabilistic solution to detect execution errors with a very high likelihood. It is based on coded processing. Section III introduced the important properties for the encoding parameter A to achieve a very high detection probability. In terms of the ISO 26262, experiments and analysis have shown that state-of-the-art diversified encoding reaches a high diagnostic coverage sufficient for the highest safety level of the ISO 26262.

Diversified encoding covers the detection of execution errors in all calculations of the safety function, i.e. in its data flow and in the state of the safety function (memory where the variables of the safety function are stored). Input and output can be covered with the solutions introduced in Section IV.D and Section IV.E.

Timing errors, e.g. a slow execution speed or a crash of the safety function, cannot be detected with diversified encoding alone. Section IV.E showed how to connect a diversified encoded safety function with a watchdog to detect and handle such timing problems reliably.

Systematic software errors, i.e. software bugs, are not covered by diversified encoding. The encoding technology and the code transformation tool have no information about a specification of a given safety function.

Direct hardware accesses, e.g. via assembler code, are outside the scope of an automatic code transformation tool. It is possible to integrate assembler code with the help of manual encoding. In such cases, it can make sense to use a mix of automatic code transformation and manual encoding.

B. *Advantages of Diversified Encoding*

The decision to use diversified encoding for detecting execution errors is a design decision that impacts sensitive parts of a system architecture and of the development process. To get the best results, diversified encoding has to be considered at an early stage in the development process.

Diversified encoding works hardware independent. It makes no assumptions on the hardware. The only requirement is that there exists a standard C compiler for the target hardware. The target hardware does not need to be certified or developed for safety-critical systems. It is even possible to use consumer-graded hardware together with diversified encoding to develop a safety-critical system.

Because diversified encoding is a software solution, it is more flexible than a hardware solution. Diversified encoding can be restricted to the safety-critical parts of a given system and puts no restrictions on the non-safety-critical system parts. An automated code transformation tool further increases the flexibility and frees development resources that can be assigned to developing the safety function itself.

The coded processing is the base of the diversified encoding. It is possible to use coded processing without diversified encoding. However, diversified encoding has two advantages over coded processing:

- Diversified encoding with AN encoding detects the same symptoms then one can detect with the more complex encodings ANB and ANBD without diversified encoding. Because of their complexities, ANB and ANBD require more computing resources than AN with diversified encoding. In other words, diversified encoding requires a lower amount of computing resources than coded processing on a comparable safety level.
- Diversified encoding detects tool errors. The code transformation tool generates the encoded safety function. Hence, its tool criticality is comparable to a compiler. However, because the diversified encoding compares the output of the native safety function with the output of the encoded safety function, it detects any encoding error that alters the functionality of the encoded safety function.

Especially when using the automated code transformation tool, the diversified encoding reduces the development effort for the final system. In addition, the diversified encoding makes cyclic memory checks and periodic instruction set tests for the safety function obsolete. Inverse storage of variables does not need to be applied to variables stored within the safety function. By avoiding these defensive programming methods, valuable development resources are released by the use of coded processing and diversified encoding. And, because these programming methods also consume system resources at runtime, their avoidance releases hardware resources of the safety-critical system.

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

References

[1] P. Forin. Vital coded microprocessor principles and application for various transit systems. In IFA-GCCT, pages 79-84, Sept 1989.

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

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

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

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