Douglas Cunningham, Petra Hofstedt, Klaus Meer, Ingo Schmitt (Hrsg.): INFORMATIK 2015
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2015

Safety Functions on Commodity Hardware with Diversified Encoding

Martin Süßkraut[1], André Schmitt[2] and Jörg Kaienburg[3]

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. One solution to this problem is to use hardware that is not certified for safety-critical systems, for example consumer-graded hardware, but that fulfills the feature and performance 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, software coded processing, software diversification

1 Introduction

Traditionally, safety-critical hardware is designed for detecting execution errors. Execution errors are errors that falsify the execution of a correct software. Examples are transient bit-flips in memory, CPU or on the BUS and permanent errors like stuck-at 0 or 1 on gates in the ALU of the CPU. 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 1, Section 2 presents an example which will be used during the paper. Sections 3 and 4 introduce software coded processing and diversified encoding, respectively.

ISO 26262 references coded processing based on Forin [Fo89] in Volume 5 under D 2.3.6 as a technology to detect hardware errors [IS11]. Forin is also the basis for software coded

[1] SIListra Systems GmbH, Königsbrückerstr. 124, 01099, Dresden, martin.suesskraut@silistra-systems.com
[2] SIListra Systems GmbH, Königsbrückerstr. 124, 01099, Dresden, andre.schmitt@silistra-systems.com
[3] SIListra Systems GmbH, Königsbrückerstr. 124, 01099, Dresden, joerg.kaienburg@silistra-systems.com

processing introduced in Section 3 [SSF09]. Software coded processing in combination with diversified encoding is a flexible solution to detect execution error with the potential to fulfill ASIL-D. ASIL-D is the highest safety level of the ISO 26262. 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 5 discusses a code transformation tool for generating the diversified encoding solution. To achieve this, the tool takes the source code of 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 the transformation tool. Previous approaches using coded processing do not have these properties.

The paper concludes in Section 6 with a discussion of the advantages of the diversified encoding approach: hardware-independence, flexibility of software and tool support for the C programming language. Hence, development and innovation cycles could be expedited. As one of the intended side effects, diversified encoding decouples hardware-error detection and functionality. The presented technique of diversified encoding with coded processing allows system architectures equipped with safety functions while running on non-certified hardware.

2 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 counter, that can be used as a reference to check a sequence of received messages. It has the following specification:

- The counter is one 32-bit counter unsigned integer value.
- The initial counter value is 1.
- The counter has an interface to query the current counter value and to increment the counter value by a given unsigned integer value.
- Execution errors while executing the 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 a C interface from this specification:
`uint32_t incSafeCounter(uint32_t incValue).`

The implementation itself does not contain any specific measures to detect execution errors. Sections 3 and 4 demonstrate how to automatically generate the source code to detect execution errors. Especially Section 4.1 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[4]):

```
1  /* include standard C library for fixed-sized integer types */
2  #include <stdint.h>
3
4  /* the safety critical counter variable */
5  static uint32_t safeCounter = 1;
6
7  /* interface of the counter */
8  uint32_t incSafeCounter(uint32_t incValue) {
9    safeCounter += incValue;
10   return safeCounter;
11 }
```

3 Software Coded Processing

Diversified encoding builds on the coded processing technology [Fo89]. 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. The data flow of a program is the sum of all computations and data processing in the program. In other words, all constants, variables and operations have to be re-programmed while making use of SCP.

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 [SSF09]. 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 a 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.

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

- 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 4.2). 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 2 with a simplified AN encoding for $A = 7$:

```
1  /* include standard C library for fixed-sized integer types */
2  #include <stdint.h>
3
4  /* 1 is AN encoded 7 */
5  static uint64_t safeCounter = 7;
6
7  uint64_t incSafeCounter_encoded(uint64_t incValue)
8  {
9    safeCounter = add_encoded(safeCounter, incValue);
10   return safeCounter;
11 }
```

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. The encoded version of the largest 32-bit unsigned integer value does not fit into 32-bit[5].

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. Hence, encoded operations must also implement over- and underflows, where they are defined by the C-standard. 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

[5] The largest 32-bit unsigned integer value is $4,294,967,295$. Encoded with $A = 7$ this value is $30,064,771,065$.

word into another valid word is roughly $\frac{2^n}{2^{n+k}} = \frac{1}{2^k}$. The probability $\frac{1}{2^k}$ 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.

4 Diversified Encoding

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. This execution only changes native state. The result of this execution is the native output.

Encoded Execution: The encoded execution is based on the encoded variant of the safety function. This 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 4.2 shows an example for such a check. If these checksums differ, an execution error has happened. A standard checksum algorithm can be used for computing the checksum, e.g. CRC32.

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. 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. The result of step ① is the encoded input values for step ②. The diversity framework uses

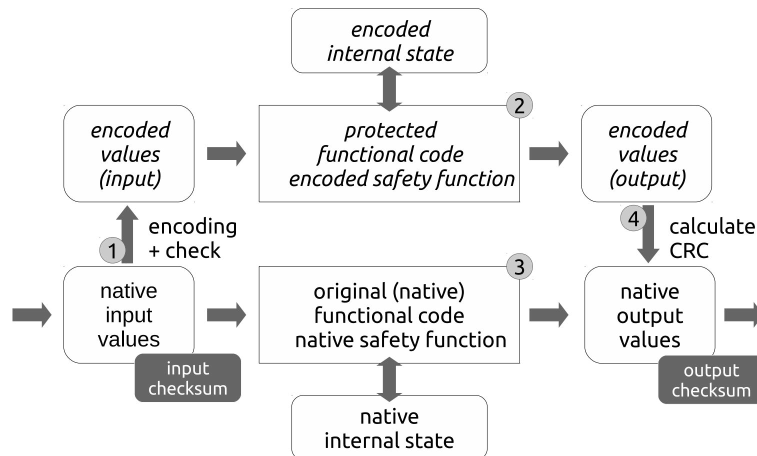

Fig. 1: Data-flow model of diversified encoding.

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. This 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 F_N. Function F_N reads the native input values. F_N operates on 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 solution detects transient execution errors because of the two executions. Permanent errors are detected, because of the diversity between the two executions. The different data representations between the two executions enable to detect freedom of interference errors

with respect to memory as execution errors. Timing errors can be detected with a watchdog (see Section 4.2). For a more detailed the safety argument see [SSK15].

4.1 Diversified Counter

This section explains the usage of diversified encoding with the example from Section 2. The code transformation tool generates an entry point function `incSafeCounterSafe` 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 `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[6]. 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 the function:

```
uint32_t SIListra_diversity_output_checksum(uint32_t returnValue).
```

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 watchdog.

4.2 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.

[6] The `checksum` variable can also be global variable, if pointer usage is discouraged.

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.

```
1  /* constant for calculating the response for the counter */
2  #define SAFE_COUNTER_SIG   (12u)
3
4  /* stores the current watchdog challenge */
5  extern uint32_t challenge;
6  /* stores the response for the watchdog */
7  extern uint32_t response;
8
9  /* initialize checksum variable with unlikely checksum value */
10 uint32_t checksum = 0;
11
12 /* increment the counter by 1 */
13 uint32_t currentCounterValue = incSafeCounterSafe(&checksum, 1u);
14
15 /* calculate response from challenge and output checksum */
16 response = challenge + checksum + SAFE_COUNTER_SIG -
17     SIListra_diversity_output_checksum(currentCounterValue);
18 /* ... */
```

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

If the values of `checksum` and `SIListra_diversity_output_checksum(currentCounterValue)` differ, then an execution error happend. In this case the value of `response` will deviate 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 `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.

5 Code Transformation

The code transformation tool generates automatically the source code of the encoded program and the source code of the diversity framework. The diversified encoding solution safe guards against tool errors, which positively influences tool classification. 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, 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 in the shown order:

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.

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.

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

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

The encoding of floating point arithmetic is still a research topic.

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 addi-

tion, 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.

6 Discussion and Conclusion

This section concludes the paper with a discussion of the current limits of diversified encoding and its advantages.

6.1 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 3 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. For instance, in [Gh15] we show that only 0.002% of 300.000.000 injected errors where not detected with the diversified encoding solution.

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 4.

Timing errors, e.g. a slow execution speed or a crash of the safety function, cannot be detected with diversified encoding alone. Section 4.2 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.

6.2 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 [Sc10a] without diversified encoding. Because of their complexities, ANB and ANBD require more computing resources than AN with diversified encoding [Sc10b]. 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. However, because the diversified encoding compares the output of the native safety function with the output of the encoded safety function, it detects errors of the code transformation tool that alter the functionality of the encoded safety function. Hence, the tool criticality is below the tool criticality of a compiler.

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. Di-

versified 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

[Fo89] Forin, P.: Vital Coded Microprocessor Principles and Application for Various Transit Systems. In: IFA-GCCT. pp. 79–84, Sept 1989.

[Gh15] Ghadhab, Majdi; Kaienburg, Jörg; Süßkraut, Martin; Fetzer, Christof: Is Software Coded Processing an Answer to the Execution Integrity Challenge of Current and Future Automotive Software-Intensive Applications? In: Proceedings of AMAA 2015, 19th International Conference on Advanced Microsystems for Automotive Applications. July 2015.

[IS11] ISO TC22/SC3/WG16: ISO/DIS 26262 - Road vehicles – Functional safety. Technical report, Geneva, Switzerland, July 2011.

[Sc10a] Schiffel, Ute; Schmitt, André; Süßkraut, Martin; Fetzer, Christof: ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In (Schoitsch, Erwin, ed.): Computer Safety, Reliability, and Security. volume 6351 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 169–182, 2010.

[Sc10b] Schiffel, Ute; Schmitt, André; Süßkraut, Martin; Fetzer, Christof: Software-Implemented Hardware Error Detection: Costs and Gains. In: The Third International Conference on Dependability (DEPEND 2010). IEEE Computer Society, Los Alamitos, CA, USA, pp. 51–57, 2010.

[SSF09] Schiffel, Ute; Süßkraut, Martin; Fetzer, Christof: 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, Berlin, Heidelberg, pp. 283–296, 2009.

[SSK15] Süßkraut, Martin; Schmitt, André; Kaienburg, Jörg: Safe Program Execution with Diversified Encoding. In: Proceedings of the 13th embedded world conference 2015. February 2015.