

White paper: Encoding Compiler and Encoded Processing

Martin Süßkraut, Ute Schiffel, André Schmitt, and Christof Fetzer

Computer Science Department
Technische Universität Dresden
Dresden, Germany

February 27, 2011

1 Introduction

In the future, decreasing feature sizes of integrated circuits will lead to less reliable hardware [3]. Currently used hardware-based solutions to detect hardware errors are expensive and usually an order of magnitude slower than commodity hardware [2]. Thus, due to economic pressure, more and more critical systems need to be based on unreliable commodity hardware. However, commodity hardware not only exhibits fail-stop behavior but also more difficult to detect and to mask *silent data corruptions* (SDCs), i.e., commodity systems will also generate erroneous output instead of crashing.

SDCs are transient or permanent. Double- or triple-modular redundancy can detect transient corruptions. However, such redundancy requires a more complicated and expensive hardware setup compared to our solution. Permanent corruptions cannot be detected by double- or triple-modular redundancy. However, in the future, hardware aging will lead to more permanent corruptions [3].

Our vision is to enable the use of cost effective unreliable commodity hardware in safety critical systems. To achieve our vision, we extend the limited failure detection capabilities of commodity hardware with the help of software. In addition to a more sophisticated failure detection, system architects can apply well known toleration approaches to mask SDCs. Our approach works well with retries, fail-over, and graceful degradation.

In this paper we introduce our *Encoding Compiler* and its underlying technology *Encoded Processing*. Applications compiled with our Encoding Compiler turn SDCs into easier to handle stop failures at runtime – without the need for custom hardware.

The upcoming automotive safety standard (ISO 26262) mentions Encoded Processing explicitly as one technique to achieve the highest safety level (ASIL D) [5]. In addition to the automotive industry, system architects working in any other safety critical application area, for instance, railway, aerospace, and medical devices, can profit from using our Encoding Compiler. Any application where the risk of erroneous executions is unacceptable high can apply our Encoding Compiler and the principles of Encoded Processing in software to reduce the risk introduced by SDCs to an acceptable level.

In summary the advantages of our Encoding Compiler are:

Safety (Resilience against hardware errors) Applications compiled with the Encoding Compiler detect influences of hardware errors that disturb their correct execution with a very high likelihood. Errors can happen anywhere in the hardware. Encoded applications detect errors modifying data during storage and transport and also processing errors, that is, errors disturbing the computation.

Safety (Resilience against some bugs in 3rd-party software) Encoded applications detect errors of 3rd-party software that influences the correct execution of the encoded application. For instance, Encoded Processing detects if the OS or another process erroneously changes the data segment or the code segment of the encoded application. However, Encoded Processing does only detect that if the application is not executed correctly, i.e., our approach does *not* detect if the application itself contains bugs. Bug identification is outside of the scope of Encoded Processing.

Simple Integration into Development Work Flow Our Encoding Compiler applies the principles of Encoded Processing as outlined in the rest of this paper. Currently, our Encoding Compiler receives C code as input and generates C code as output. In that way it can be transparently inserted into existing tool chains. The Encoding Compiler additionally can directly generate x86_64 code. We also consider to directly support more hardware platforms in the future.

Support for Distributed Systems System architects can place the detection on the same hardware node which runs the encoded application. However, to increase safety architects can also choose to add a separate watchdog node, which continuously monitors the application running on the commodity hardware node. The watchdog can also detect timing failures.

In distributed systems, the destination node of a message, which was produced by an encoded application, can check if the message was calculated correctly. Furthermore, if a distributed encoded application runs on several nodes it can safely exchange messages between the nodes it runs on. Any erroneous manipulation of messages, by the OS or hardware failures in the transmission process will be detected with a high likelihood.

Adaptable Scope of Application Depending on the requirements of a system, a system architect can choose to apply Encoded Processing to the whole application or only to some components. This is useful in mixed-mode environments where safety crit-

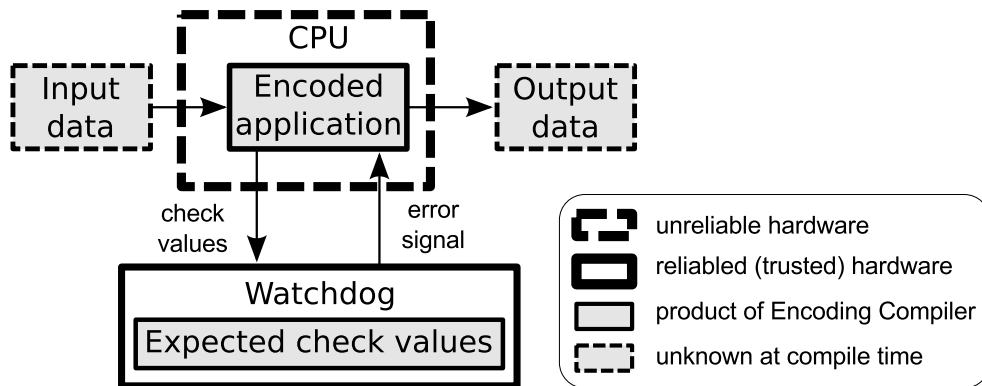


Figure 1: Encoded applications can run on unreliable hardware. Beside output it produces check values in regular intervals. A watchdog continuously monitors these check values and compares them with pre-calculated values. If the check values received by the watchdog do not match the expected check values or if the watchdog does not receive the check values in a time, the watchdog sends an error signal that aborts the application. Thereby, SDCs are prevented and instead crashes are generated.

ical components run alongside with non-safety critical components. In the following, when we refer to applying Encoded Processing to applications, we implicitly include the possibility to applying Encoded Processing only to some but not all sub-components of an application.

Adaptable Safety Our Encoded Compiler supports three different levels of encoding with increasing safety, i.e., the probability that a SDC remains undetected shrinks with each level. As expected, our measurements show that the higher the safety level the more CPU cycles are needed. However, the amount of CPU cycles increases *linearly* but the probability of detecting errors increases *exponentially*.

In the rest of this paper we describe the application and components of our Encoding Compiler and the principles of Encoded Processing.

2 Encoding Compiler

Our Encoding Compiler transforms a given application by applying the principles of Encoded Processing. We call the input of the Encoding Compiler *original application* and the output *encoded application*. The two goals of the transformation are:

Correctness If no error influences the correct execution of the encoded application and the original application, then both application should behave the same. This also means, if the original application contains a software bug, the encoded application contains this bug too.

Safety Any erroneous change of the data computation, the data flow or the control flow during the execution of an encoded application is detected with high probability.

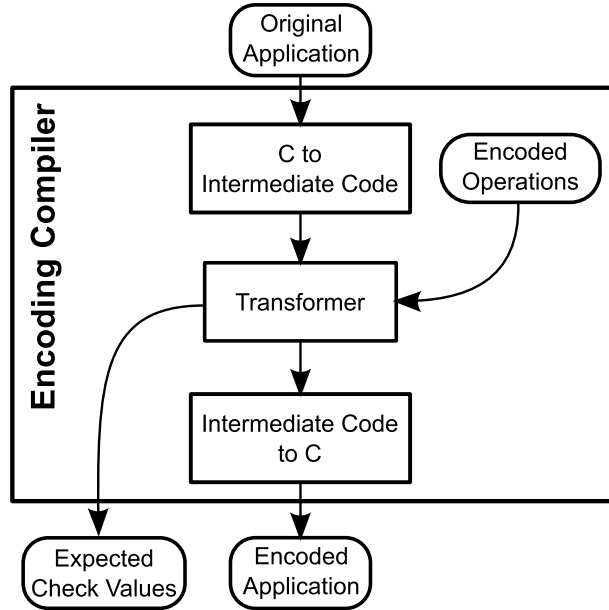


Figure 2: Our Encoding Compiler is a C-to-C transformer. Its core is the transformer that operates on an intermediate code. Hence, first the Encoding Compiler’s frontend translates the original application into the intermediate code. Then, the transformer applies the principles of Encoded Processing to this intermediate code. Together with the transformed intermediate code, the transformer outputs the expected check values, which the watchdog uses to check the execution of the encoded application. Last, the backend of the Encoding Compiler translates the encoded intermediate code back to C-code.

Currently, we focus on hardware failures and erroneous behavior of 3rd-party components and exclude malicious manipulations by attackers.

Figure 1 shows one possible architecture for running encoded applications. The encoded application and the *expected check values* are the products of our Encoding Compiler. At runtime the encoded application continuously produces check values. These check values have the following properties:

- They are independent from the concrete data processed by the encoded application.
- All computations, data flows, and control flows of the encoded application influence the check values sent. Hence, with high likelihood any erroneous manipulation of the encoded application’s execution, for example, caused by an hardware error, also influences the produced check values.
- The Encoding Compiler pre-calculates the sequence of check values that the encoded application will produce if the application is executed correctly, that is, without errors.

By comparing the check values received from the encoded application with the pre-calculated (expected) check values the watchdog monitors if the encoded application is executed cor-

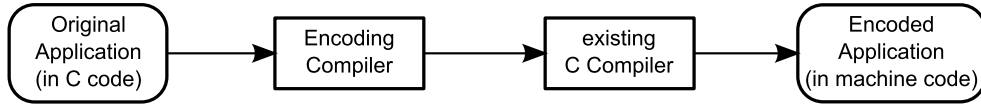


Figure 3: Our Encoding Compiler can be easily inserted into an existing build process. It just needs to be placed in front of the existing C compiler.

rectly (see Section 2.4). For additional safety, the output data can also carry check values. We call this data *encoded data*. In a distributed setting, the destination node that receives the encoded output data can verify that the output is a result of a correct execution of the encoded application. Section 2.3 gives more details about checking output data.

Our Encoding Compiler consists of two major components as depicted in Figure 2:

Encoded Operations We provide encoded versions of all operations that an application might use. These we call *encoded operations* (see Section 2.2).

Transformer The transformer operates on intermediate code. It substitutes all operations in the original unencoded application with their corresponding encoded operation. Additionally, it pre-calculates the expected check values and inserts code to send out the check values to the watchdog at runtime (see Section 2.5).

Additionally, the Encoding Compiler contains components to translate C-code into the intermediate code used by transformer and vice versa.

The Encoding Compiler is a C-to-C transformer. Thus, it can wrap an existing C compiler as show in Figure 3.

Before we present more details about the components of the Encoding Compiler and the architecture used to execute encoded applications, we introduce the principles of Encoded Processing.

2.1 Principles of Encoded Processing

Encoded Processing approaches add redundancy to any value that is part of an application’s state. This redundancy enables us to detect erroneous executions. The Encoding Compiler adds redundancy to all data words. The redundancy added divides the domain of possible data words into valid data words and into invalid data words. Valid data words are only a small subset of all possible data words.

For adding the redundancy, our Encoding Compiler uses arithmetic codes. An application encoded by our Encoding Compiler solely processes encoded data, i.e., all inputs are valid code words of an arithmetic code and all computations use and produce encoded data. Thus, we have to use solely operations that preserve the code in the error-free case.

Figure 4 shows the relation between valid data words – also called *valid code words* – and all possible data words. Correctly executed arithmetic operations preserve the arithmetic code used for adding the redundancy. Thus, given valid code words as input, the output of a correctly executed arithmetic operation is also a valid code word. A faulty arithmetic operation or an operation called with non-code words produces a result that is an invalid code word with high probability [1]. Furthermore, arithmetic codes also detect errors

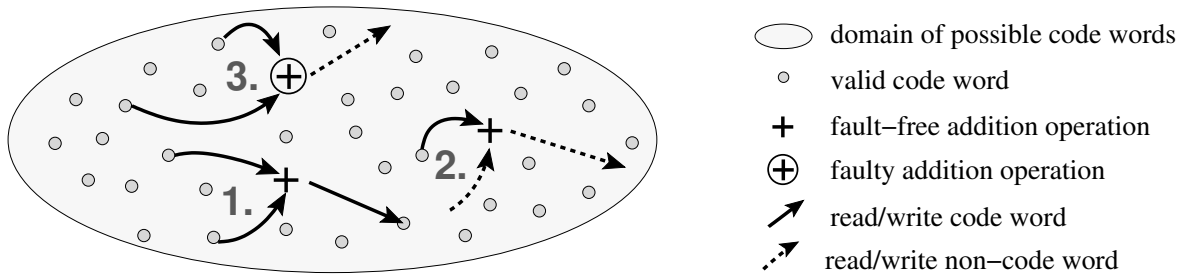


Figure 4: Our encoding adds redundancy to every variable. Of all possible data words only a small subset are valid data words.

modifying data during storage or transport because such errors most likely transform a valid code word into a non-code word.

AN-code. For an *AN-code* the encoded version x_c of variable x is obtained by multiplying its original functional value x_f with a constant A . To check the code, we compute the modulus of x_c with A , which is zero for a valid code word. As described above, all variables in a program are replaced with their encoded versions, i.e., multiples of A .

An AN-code can detect *faulty operations*, i.e., incorrectly executed operations, and *modified operands*, i.e., data that is for example hit by a bit flip. These errors are detected because they result with high probability in a data word that is not a multiple of A . The probability that such an error results in a valid code word is approximately $\frac{1}{A}$ [4].

Consider the following unencoded C code:

```

int f(int x, int y, int z) {
  int u = x + y;
  int v = u + z;
  return v;
}

```

Its AN-encoded version¹ uses solely AN-encoded data:

```

int_c f(int_c xc, int_c yc, int_c zc) {
  int_c uc = xc + yc; // uc = A*xf + A*yf
                    //      = A(xf+yf)
  int_c vc = uc + zc; // vc = A(xf+yf+zf)
  return vc;         // expected: vc mod A == 0
}

```

Yet, when a bit flip happens on the (unencoded) address bus, a wrong memory word will be accessed that with high probability contains also a multiple of A . Thus, this so-called *exchanged operand* is not detectable with an AN-code because the error is also a multiple of A . A bit flip in the instruction unit of a CPU might cause the execution of a wrong operation (*exchanged operator*) that might also not be detected by an AN-code because many operators preserve an AN-code.

¹The presented pseudo code is simplified and ignores over- and underflow issues. The comments depict the variable content in the error-free case.

ANB-Code. Forin in [4] introduced *static signatures* (so-called “*B*”s). The resulting *ANB-code* can additionally detect exchanged operator and exchanged operand errors. The encoding of a variable x in *ANB-code* is defined as $x_c = A * x_f + B_x$ where B_x is chosen for each input variable with $0 < B_x < A$. To check the code of x_c , x_c ’s modulus with A is computed. The result has to be equal to B_x that is either assigned or precomputed at encoding time.

When encoding an application, our Encoding Compiler assigns static signatures to the input variables. Knowing the program, the Encoding Compiler can pre-compute the result’s expected signature.

Now consider that there is a bit flip on the address bus when storing variable y_c . Thus, we have a *lost update* on y_c because y_c is stored in a wrong memory location. When reading y_c the next time, the old version of y_c is read – which is correctly ANB-encoded but outdated.

ANBD/ANBDmem-Code. To detect the use of outdated operands, i.e., lost updates, Forin introduced a version D that counts variable updates [4]. In the resulting *ANBD-code*, the encoded version of x is $x_c = A * x_f + B_x + D$. In Forin’s implementation, the code checker has to know the expected D to check the validity of code words.

Currently, our ANBD-code implementation does only apply versions to memory that is accessed using load and store instructions but not to registers. Therefore, the load and store instructions remove or add the version information as required. We denote this encoding as *ANBDmem-code* in the following.

Safety Levels Our Encoding Compiler supports all three presented arithmetic codes. Since the ANBDmem-code detects more failure classes than the ANB-code, which detects more failures than the AN-code, we use ANBDmem-encoding for the highest safety level and AN-encoding for the lowest safety level. Note that our measurements have shown that AN-encoding detects already more errors than double-modular redundancy.

2.2 Encoded Operations

Most of the normal operations of the intermediate code do not preserve the arithmetic encoding even when there is no hardware error. Consider the multiplication of two AN-encoded variables x_c and y_c : $x_c * y_c = A * x * A * y$. The result must be corrected with a division by A . The concrete correction operation depends on the specific arithmetic code used and the operation executed. Thus, we have to provide a so-called *encoded operation* for each operation of the intermediate code.

Each encoded operation is implemented in three versions for AN-, ANB-, and ANBDmem-encoding. Each version contains the appropriate correction operations. We hand-implemented all encoded operations. We took great care to ensure

- that a correct execution of an encoded operation produces a correct result that is a valid code word and

Supported Operations	Examples
Arithmetic integer operations	<code>+, - * / rem</code>
Bitwise logical integer operations	<code>and, or , xor</code>
Shift Operations	<code>shl, shr, ashr</code>
Integer Comparison	<code>cmp eq, cmp ne, cmp gt, cmp ge, cmp lt, cmp le</code>
Memory Operations	<code>load, store, alloca, malloc, free</code>
Conversion Operation	<code>trunk, zext, sext, ptrtoint, inttoptr</code>
Pointer Arithmetic	<code>getelementptr</code>

Table 1: The Encoded Operations library supports these operations.

- that any erroneous execution of an encoded operation or its execution with non-code words as parameters results in a non-code word with high probability.

Table 1 gives an overview of the operations that we support in our library of encoded operations. Floating pointer operations are not directly supported. However, we support floating operations implemented on top of integer operations. In comparison to previous approaches [4], we support the full range of integer operations and dynamic as well as static memory management.

2.3 Interfaces

Figure 1 shows the input data which an encoded application processes and the output data which an encoded application produces. Input data is unencoded if it is not provided by another encoded application. That means, input data need to be encoded with the encoding scheme of the application (AN-, ANB-, or ANBDmem-encoding).

The encoded application produces encoded output data. If the output data is not used by another encoded application as input, it has to be decoded. Part of the decoding step is to check if the output data is a valid code word.

The encoding and decoding of input data and output data, respectively, is not protected by Encoded Processing. These steps can be protected by well-know approaches like triple module redundancy. For example, the input can be encoded three times and the already encoded input can then be compared within the encoded application. The comparison step is already protected by Encoded Processing.

To provide maximal safety, the Encoding Compiler needs access to all 3rd-party code that is used by the encoded application. Where this is not possible (for instance the OS) any data passed to 3rd-party components has to be decoded and data returned from 3rd-party components has to be encoded. In this case the processing of data by 3rd-party components is not protected by Encoded Processing. However, there are exceptions. For instance the file-system: If the files are only read by the same or other encoded applications it is possible to write encoded data directly to disk. Hence, the data is protected against erroneous manipulations of the file system stack (libraries, OS) and storage hardware (data bus, hard disk) by Encoded Processing. The same approach works for the network.

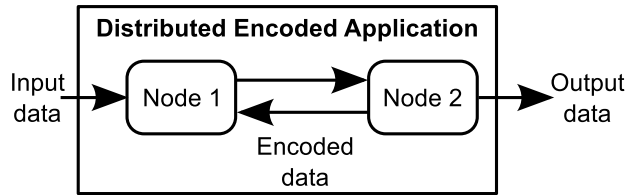


Figure 5: Example for a distributed encoded application: Encoded data can be sent between the nodes executing encoded applications without any additional decoding and encoding. Thus, also during the transmission the data sent is protected by Encoded Processing.

When in a distributed encoded application, a node sends data to another node, the first node can directly pass encoded data to the network stack to send it to the second (also encoded) node. The second node reads this data directly from the network stack without encoding. Figure 5 illustrates this approach with a distributed application consisting of two nodes. Of course, any input and output to the distributed application itself has to be encoded and decoded, respectively, as we have described previously.

2.4 Validity Checks

We already discussed all the possibilities to check the correctness of the execution of an encoded application throughout the paper. With this subsection we want to briefly summarize them in one place.

There are two basic ways to verify the correct execution:

Checking output data Whenever data needs to be decoded, we also check that the data only consists of valid code words. This check itself is not protected by Encoded Processing. We can protect the decoding by well-known approaches like triple modular redundancy.

Another possibility is to offload the decoding into a small separate and reliable hardware. The still encoded output data is sent to this hardware and is decoded and checked on the reliable hardware only. This reliable hardware component can be merged with the watchdog (see below).

In any case the goal is to only output data that results from a correct execution of the encoded application.

Intermediate checks Intermediate checks are an optimization. They detect erroneous executions independent of operations performing output by checking the state of the execution. Hence, recovery mechanisms can be used earlier. Like the checking of output data, the checking itself is not protected by Encoded Processing. Additional protection is not necessarily needed because the decoding code always has to check the output data.

Intermediate checks can be placed at three locations:

Embedded in the encoded application The transformer can embed checking code

into the encoded application. However, this code is executed on the same unreliable hardware. Hence, these checks are just an optimization.

Watchdog A watchdog (see Figure 1) continuously monitors the execution of the encoded application. Therefore, the transformer inserts code into the application to send check values to the watchdog.

These a check values capture the complete execution since the last check value sent. Thus, with a high probability any error happening between to adjacent sent operations invalidates the check value.

Receiver node In distributed systems a node receiving encoded data could (but not needs to) check the validity of this data. For instance, Node 2 in Figure 5 could check any data received from Node 1.

2.5 Transformer

The transformer receives the original application in an intermediate code format. Its output is the encoded application in the same intermediate format, a sequence of pre-computed check values, and the pre-computed signatures for the output data.

The goal of the transformation step is that any error disturbing the execution of the encoded application changes the check values sent out by the encoded application with a high probability *and* invalidates the output data. Hence, execution errors can be detected by the approaches listed in Section 2.4.

To achieve this goal the transformer has two tasks:

Data flow & computation The transformer extends every variable with redundancy using arithmetic encoding. For ANB- and ANBDMem-encoding, the transformer also assigns signatures to every variable. Furthermore, it encodes the initialization values of global variables and constants. The transformer replaces any operation listed in Table 1 with its encoded version from our encoded operations library.

Control flow Additionally every control flow operation is instrumented to protect it against errors resulting in an erroneous modification of the instruction pointer. The transformer directly supports unconditional jumps, conditional jumps and function calls. The C to intermediate code step translates high-level constructs such as `switch` and any loops into jumps. Thus, these high-level constructs are also supported.

Check values Last but not least, the transformer inserts code to calculate the check values. These check values combine checking of data computation, data flow and control flow and are calculated in a way to match the pre-computed expected check values at runtime.

Note that the encoded intermediate code produced by the transformer is already protected against any errors happening in the next translation steps, that is, the translation from intermediate code to C, the compilation of this C code, storage of the program etc.

3 Conclusion

Our Encoding Compiler is an effective approach to protect a C application against execution failures. Because our Encoding Compiler is a C-to-C translator it can easily be integrated into existing build processes.

We have evaluated our Encoding Compiler on x86 systems. In our measurements ANB- and ANBDMem-encoded applications detect in average more than 99% of the SDCs that the original, unprotected application produced when exposed to errors. Note that this explicitly excludes the hardware errors that already led to fail-stop behavior in the original application. For AN-encoded applications the detection rate is more then 90% of the SDCs not detected by unprotected applications.

The additional code inserted by the transformer reduces the throughput of the encoded applications compared to the original application. In our measurements we demonstrated that the encoded applications have an average throughput in the range of 50% and 70% of the throughput of the original application.

For the future, we plan to further optimize our Encoding Compiler for more speed and more safety of the encoded applications. And we plan to evaluate its usage in the settings of embedded systems.

4 Acknowledgements

We like to thank German Federal Ministry of Economics and Technology and EXIST for supporting the SIListra project.

Gefördert durch:



Bundesministerium
für Wirtschaft
und Technologie

aufgrund eines Beschlusses
des Deutschen Bundestages



References

- [1] A. Avizienis. Arithmetic error codes: Cost and effectiveness studies for application in digital system design. In *Transactions on Computers*, 1971.
- [2] Hugh J. Barnaby. Will radiation-hardening-by-design (RHBD) work? *Nuclear and Plasma Sciences, Society News*, 2005.
- [3] Shekhar Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 2005.

- [4] P. Forin. Vital coded microprocessor principles and application for various transit systems. In *IFA-GCCT*, pages 79–84, Sept 1989.
- [5] ISO/IEC. *Draft BS ISO 26262-5:2009 Road vehicles – Functional safety – Part 5: Product development:hardware level*. ISO, Geneva, Switzerland.