ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software

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

Technische Universität Dresden
Department of Computer Science
<http://wwwse.inf.tu-dresden.de>
Dresden, Germany
{ute, andre, suesskraut, christof}@se.inf.tu-dresden.de

Abstract. It is expected that commodity hardware is becoming less reliable because of the continuously decreasing feature sizes of integrated circuits. Nevertheless, more and more commodity hardware with insufficient error detection is used in critical applications. One possible solution is to detect hardware errors in software using arithmetic AN-codes. These codes detect hardware errors independent of the actual failure modes of the underlying hardware. However, measurements have shown that AN-codes still exhibit large rates of undetected silent data corruptions (SDC). These high rates of undetected SDCs are caused by the insufficient protection of control and data flow through AN-codes. In contrast, ANB- and ANBD-codes promise much higher error detection rates because they also detect errors in control and data flow. We present our encoding compiler that automatically applies either an AN-, ANB- or ANBD-code to an application. Our error injections show that AN-, ANB-, and ANBD-codes successfully detect errors and more important that indeed ANB- and ANBD-codes reduce the SDC rate more effectively than AN-codes. The difference between ANBD- and ANB-codes is also visible but less pronounced.

1 Introduction

In the future, decreasing feature sizes of integrated circuits will lead to less reliable hardware [6]. Currently used hardware-based solutions to detect hardware errors are expensive and usually an order of magnitude slower than commodity hardware [3]. Thus, due to economic pressure, more and more critical systems will 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., they generate erroneous output instead of crashing. To use this unreliable hardware in critical systems, it is required to extend its limited failure detection capabilities with the help of software. We implemented a system that turns SDCs into much easier to handle stop failures – without the need for custom hardware.

When implementing detection of hardware errors in software, more CPU cycles are needed to execute an application. However, instead of custom reliable, commodity hardware can be used. Typically, commodity hardware is not only cheaper than custom reliable hardware but also faster because it uses the newest hardware components. Furthermore, in many systems, only a few application components are critical and only these components need to be protected by additional error detection. Hence, we can bound the performance impact of software-based error detection by focusing on critical application components.

Our error detection approach is based on *arithmetic codes* (see Sec. 2) that support end-to-end software-implemented hardware error detection, i.e., protect data from undetected errors during storage, transport, and computations as well. Their error detection capabilities are decoupled from the actual hardware.

For using arithmetic codes, it is required to enable programs to cope with arithmetically encoded data. Therefore, we developed our *encoding compiler* that supports different arithmetic codes that we will introduce in Section 2: 1. AN-code, 2. ANB-code, and 3. ANBDmem-code. These codes provide different error detection rates at different runtime costs. Thus, systems engineers can balance gain and costs.

This paper presents the extension of our AN-encoding compiler presented in [14] with support for ANB- and ANBDmem-encoding. In contrast to the AN-code, the newly added ANB- and ANBDmem-code facilitate also the detection of data and control flow errors. While ANB-encoding of arithmetic operations was already presented in [17], this paper focuses on ANB/ANBDmem-encoding of control and data flow (see Sec. 3). In contrast to existing solutions such as [8], our encoding compiler supports arbitrary control and data flow that is not predictable at encoding, i.e., compile, time. Our evaluation (see Sec. 4) shows that indeed the amount of SDCs for ANB- and ANBDmem-encoded programs compared to unencoded programs goes down by 99.2% and 99.7% respectively. In contrast, AN-encoding leads only to a reduction by 93.5%. Furthermore, we show that compiler-based ANB/ANBDmem-encoding induces much less runtime overhead than our previously presented interpreter-based approach [18] that not even was as complete as the encoding compiler is.

2 Arithmetic Codes

Arithmetic codes are a technique to detect hardware errors during runtime. The encoding adds redundancy to all data words. Valid code words are only a small subset of all possible data words.

Correctly executed arithmetic operations preserve the code, i.e., given valid code words as input, the output is also a valid code word. A faulty arithmetic operation or an operation called with non-code words with high probability produces a result which is an invalid code word [2]. Furthermore, arithmetic codes also detect errors modifying data during storage or transport.

When an application is encoded using an arithmetic code, it will solely process encoded data, i.e., all inputs have to be encoded and all computations use

and produce encoded data. Thus, we have to use solely operations that preserve the code in the error-free case.

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.

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 in data that is not a multiple of A with high probability. The probability that such an error results in a valid code word is approximately $\frac{1}{A}$ [8]. 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.

ANB-Code. Forin in [8] 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.

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 ANB-encoded version¹ uses solely ANB-encoded data:

```
int_c f(int_c xc, int_c yc, int_c zc) {
    int_c uc = xc + yc;    // uc = A*x_f+B_x + A*y_f+B_y
                          //      = A(x_f+y_f)+B_x+B_y
    int_c vc = uc + zc;    // vc = A(x_f+y_f+z_f)+B_x+B_y+B_z
    return vc;             // expected: vc mod A == B_x+B_y+B_z
}
```

When encoding the program **f**, we assign static signatures to the input variables **x**, **y**, and **z**. Knowing the program, we can precompute the result’s expected signature $B_v = B_x + B_y + B_z$. Note that for implementing dynamically allocated memory, we use dynamic signatures that we introduced in [18]. These are assigned at runtime. If now an error exchanges variable y_c with another encoded variable $u_c = A * u_f + B_u$, the result’s computed signature $v_c \bmod A$ would be $(B_x + B_u + B_z)$ instead of the precomputed, i.e., expected, $(B_x + B_y + B_z)$. If the first addition is erroneously replaced by a subtraction, the resulting computed signature is $(B_x - B_y + B_z)$ instead of $(B_x + B_y + B_z)$. Thus, an ANB-code can

¹ The presented pseudo code is simplified and ignores the over- and underflow issues described in [17]. The comments depict the variable content in the error-free case.

detect exchanged operands and operators additional to faulty operations and modified operands. However, 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 [8]. In the resulting *ANBD-code*, the encoded version of x is $x_c = A * x_f + B_x + D$. 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. Thus we denote it as *ANBDmem-code* in the following.

3 Encoding an Application

Encoding an application, i.e., enabling it to process encoded data, can be done at different stages of an application’s lifetime: before compilation by encoding the source code, during compilation by encoding an intermediate representation of the program, or at runtime by encoding the binary during execution.

Forin’s *Vital Coded Processor (VCP)* [8] ANBD-encodes an application on source code level. As we pointed out in [19], VCP requires knowledge of the complete data and control flow of the encoded program to precompute the signatures of all output variables for code checking. This prohibits the usage of dynamically allocated memory and function pointers. Furthermore, encoding loops and nested control flow structures at source code level is cumbersome and not described by Forin. Forin presents neither an evaluation of the error detection capability of VCP nor any runtime measurements.

Software Encoded Processing (SEP) introduced by us in [18] implements ANBD-encoding on assembler level at runtime. Therefore, we developed an interpreter for programs given as binary that itself is encoded using the principles of VCP [8]. Thus, we can encode arbitrary programs with arbitrary control flow. To encode dynamically allocated memory, *dynamic signatures* that are determined at runtime were introduced. The error injection results presented in [18] show that SEP successfully prevents erroneous output. However, the observed slowdowns make SEP unusable in practice.

In this paper, we present our *compiler based encoding (CBE)*. CBE encodes programs at the intermediate code level. In our case, by instrumenting LLVM code [9]. Adding the encoding at intermediate code level at compile time needs new concepts to encode the control flow. However, it makes encoding control flow easier compared to VCP because we have not to handle nested control structures explicitly. In contrast to the VCP, CBE provides support for programs with arbitrarily nested control structures and dynamically allocated memory. Furthermore, all programming languages for which an LLVM compiler exists can be supported. So far, we tested our implementation for C programs.

In contrast to SEP, CBE provides a more complete protection because: 1. it also encodes bitwise logical operations and floating point operations not covered by SEP, and 2. it also protects against bugs in the compiler back-end that generates code for a specific machine. At the same time, CBE introduces much less overhead than SEP because no expensive interpretation is required. Furthermore, CBE restricts usage of expensive dynamic signatures to dynamically allocated memory. CBE uses *static* signatures (i.e., computed at compile time) for all statically allocated memory. In contrast, in SEP, every data item has a dynamic signature because all signatures are assigned at runtime due to the interpreter-based implementation.

For encoding a program with an AN-, ANB-, or ANBDMem-code, every instruction and every variable has to be replaced with its appropriate encoded version. Thus, we need:

1. encoded versions of all instructions supported by LLVM,
2. to encode all constants and initialization values,
3. to handle calls to external libraries, and
4. to encode control and data flow, that is, we have to check that instructions are executed in the correct order with the right operands and that all conditional jumps are executed correctly.

(1) Encoded Instructions. How we encode basic arithmetic and boolean operations we described in [17]. Encoding more complex operations such as bitwise logical operations, type casting, shifting, or floating point operations we described in [14]. In this paper here we focus on encoding control and data. This was not yet supported by our AN-encoding compiler presented in [14].

(2) Encoding Constants and Initializers. Since we choose *A* and the static signatures at encoding time, i.e., compile time, we can replace the unencoded constants and initializers with their encoded versions at compile time.

(3) External Calls. In contrast to SEP, the static instrumentation of CBE does not allow for protection of external libraries whose source code is not available at compilation time. For calls to these libraries, we currently provide hand-coded decoding wrappers, which decode (including code check) parameters and, after executing the unencoded original, encode the obtained results. For implementing those wrappers, we rely on the specifications of the external functions.

(4) Data and Control Flow (CF). While an AN-code only detects operation and modified operand errors, we can use an ANB-code in a way that ensures also the detection of exchanged operands and operators and arbitrary combinations of these errors. The ANBDMem-code can also detect lost updates of memory.

VCP requires statically predictable control flow and allows output only at one specific point in the program execution. Only at this point execution errors are detectable because only there the code of the output is checked. In contrast, we implement for CBE a continuous checking of the program execution because 1. CBE allows output at arbitrary positions, 2. we do not know the control flow statically, and 3. CBE provides *fail-fast* behavior, that is, detects errors as fast as possible, thereby, allowing for an earlier reaction to them.

Therefore, our encoded application continuously produces check values, which it sends to a *watchdog*. The goal of the encoding is that if an execution error happens, the encoded application will not send the expected check value to the watchdog. The expected check values are statically determined and given to the watchdog as an ordered list s , which is indexed by a counter i . i counts the received check messages. The encoded application also has a counter i for sent check messages. This allows the application to provide the expected check value in an error-free run. Therefore, the application contains a list δ , which has the same size as the watchdog's list s . However, δ contains the differences of consecutive elements of s , i.e., $\delta[i] = s[i+1] - s[i]$.

We assign signatures to all input variables (parameters, memory reads, and return values of external functions) at encoding time. Using these signatures, we compute – also at encoding time – for every basic-block a *block signature* (BBx) that is the sum of the signatures of all results produced in this block.

Furthermore, we add an *accumulator* acc to the application. acc is initialized for each basic-block x so that it contains the next $s[i]$ minus the basic-block signature BBx . While the basic-block is executed, the signatures of all produced results are added to acc . At the end of the block, acc should equal $s[i]$ and is sent (**send**) to the watchdog. acc will not contain the expected value if any error modified the data flow, computations, or data. After sending acc , it is adapted for the next basic-block. Thereby, we can provide control flow checking. In contrast to existing solutions, our control flow checking provides more than inter-basic-block checking. We also check that every instruction was executed in the correct order, with the right operands, and its execution itself was error-free.

To prevent jumping from anywhere before a **send** of acc to any other **send**, we assign to each basic-block an ID BBx_id . The ID BBx_id is subtracted from acc before a block is executed and it is also sent to the watchdog. The watchdog checks if $acc + BBx_id == s[i]$. If not, the watchdog shuts down the application.

Inter-basic-block-CF and Unconditional Jumps. Consider the following example in LLVM bytecode:

```

1 bb1:
2   x = a+b
3   y = x-d
4   br bb2

```

Our ANB/ANBDmem-encoding compiler transforms this example to:

```

1 bb1: ; acc=s[i]-BB1-BB1_id  BB1=Bx+By=(Ba+Bb)+(Ba+Bb-Bd)
2   xc = addc(ac, Ba, bc, Bb) ; Bx=Ba+Bb
3   acc += xc mod A          ; acc=s[i]-BB1-BB1_id+Ba+Bb
4   yc = subc(xc, Bx, dc, Bd) ; By=Bx-Bd=Ba+Bb-Bd
5   acc += yc mod A          ; acc=s[i]-BB1-BB1_id+2*Bb-Bd
6                               ; =s[i]-BB1-BB1_id+BB1 =s[i]-BB1_id
7   send(acc, BB1_id)
8   acc += delta(i)           ; acc=s[i+1]-BB1_id
9   i++                       ; acc=s[i]-BB1_id
10  acc += BB1_id-BB2-BB2_id  ; acc=s[i]-BB2-BB2_id
11  br bb2

```

The comments (denoted by ';') show the expected value of the accumulator. Note that xc means the encoded version of x where x can be either a variable

or a function/instruction. Line 1 shows which value acc has at the beginning of $bb1$. This is ensured by the previously executed block. Lines 2 and 4 contain the encoded versions of the original instructions whose signatures are added to acc directly after executing the instructions. In line 5, acc has the value $s[i] - BB1_id$. In the next line, acc and the constant $BB1_id$ are sent to the watchdog who checks if the sum of both values equals the expected $s[i]$. The following lines adapt acc for the next basic block. Line 8 ensures that acc will contain the next check value $s[i+1]$ and line 10 adds $BB1_id - BB2 - BB2_id$. Note that this value is computed at compile time and, hence, is constant at runtime. Its addition removes this block's ID $BB1_id$ and instead introduces the next block's ID $BB2_id$ and signature $BB2$.

Conditional Jumps. Encoding conditional jumps additionally requires to check that the reached jump destination matches the actual branching condition. Consider the following example in which $cond$ is the branch condition:

```

1 bb1:
2   cond = ...
3   br cond bb_true, bb_false

```

The encoded version is:

```

1 bb1: ; acc=s[i]-BB1-BB1_id    BB1=Bcond
2   condc = ...                ; condc=A*0+Bcond if cond is false
3                               ; or A*1+Bcond if cond is true
4   acc += condc mod A          ; acc=s[i]-BB1-BB1_id+Bcond
5   send(acc, BB1_id)           ; acc=s[i]-BB1_id
6   acc += delta(i)             ; acc=s[i+1]-BB1_id
7   i++                         ; acc=s[i]-BB1_id
8   acc += BB1_id-BBtrue-BBtrue_id-(A*1+Bcond)
9                               ; acc=s[i]-BBtrue-BBtrue_id-(A*1+Bcond)
10  cond = condc / A            ; get functional value of condc
11  acc += condc                ; acc=s[i]-BBtrue-BBtrue_id-(A*1+Bcond)+condc
12  br cond bb_true, bb_false_correction
13
14 bb_true: ; condc = A*1+Bcond => acc=s[i]-BBtrue-BBtrue_id
15  ...
16
17 bb_false_correction: ; condc=A*0+Bcond=Bcond
18  ; => acc=s[i]-BBtrue-BBtrue_id-A*1
19  acc += A+BBtrue+BBtrue_id-BBfalse-BBfalse_id
20  ; => acc=s[i]-BBfalse-BBfalse_id
21  br bb_false
22
23 bb_false: ; acc=s[i]-BBfalse-BBfalse_id
24  ...

```

In line 4 acc is used to check the computation of the condition $condc$ with the already introduced approach. After sending acc , we adapt it in line 8 for the basic-block bb_true and for checking if the executed branch matches $condc$. For the latter, we subtract $A * 1 + Bcond$, the value $condc$ has if $cond$ is true. The value added in line 8 is a constant known at encoding time. In line 11, we add $condc$. If the condition is true, acc now contains the correct block signature and ID at the start of bb_true . If it is false, we have to do additional corrections which are executed in the basic-block $bb_false_correction$ before jumping to the actual destination bb_false . These corrections ensure that when bb_false is entered, acc contains bb_false 's signature and ID. If the branch in line 12 does not match

condc, *acc* will not contain the expected block signature and ID and thus a wrong check value will be sent to the watchdog. Therefore, it is required that $BBfalse + BBfalse_id \neq BBtrue + BBtrue_id$.

Function Call. For a function call, we have to validate that 1. the correct function is called, 2. with the correct unmodified parameters, and 3. the function is executed correctly.

To ensure **1.**, we assign every function a *function signature* by which it has to modify *acc*. Before the function returns, it adapts *acc* for the remainder of the calling basic-block minus this function signature. For non-void functions, an additional signature is assigned to the return value. This guarantees a predictable signature for the return value.

For ensuring **2.**, we add the expected signatures of the parameters (known at encoding time and thus constant) to *acc* before entering the function. In the function, we subtract the signatures of the actual used parameters (computed at runtime). If they do not match, *acc* will become invalid. Afterwards, the signatures of the parameters are corrected to function-specific ones which are independent of the call-site. Therefore, statically computed correction values are used that depend on the call-site and, thus, are given as constant function parameters.

Before starting executing the function, *acc* is adapted. The remaining signature and ID of the basic-block which contains the call-site is removed and the signature and ID of the first basic-block of the function are added. The used correction value is determined at encoding time and provided as constant function parameter. Thereafter, the execution continues as described before – now executing and checking the basic-blocks of the function called. These measures ensure **3.**

Watchdog. The watchdog is used to check the correct execution of the encoded program during its runtime. It is not part of the encoded program. The watchdog needs to be executed reliably outside of the encoded program.

To check the execution, the watchdog checks if the sum of the received values *acc* and basic-block ID equals $s[i]$. If the watchdog encounters an unexpected $s[i]$ or the application stops sending values (detected using a timeout), the watchdog terminates the application. If the end of *s* is reached, both application and watchdog start again at the beginning of *s* by setting *i* to zero. In improbable scenarios, this might lead to undetected errors. Yet, the more entries *s* has, the smaller is the probability of such undetected errors.

The watchdog has to iterate over *s*, do periodic comparisons with the check values received, and has to test if the application is still alive. Its easy implementation supports the application of various mechanisms to make its execution safe, e.g., redundant execution on different hardware such as onboard FPGAs or the graphics unit, or hand-encoding according to VCP [8]. Additionally, we can use multiple watchdogs in parallel to further reduce the risk of an erroneous watchdog.

Memory. Up to now we focused on values stored in registers where we use static signatures known at encoding time. Since we cannot predict memory access

patterns at encoding time, we need to use dynamic signatures, calculated at runtime, for values stored in memory. When storing a value, we convert its static signature into a dynamic signature that depends on the address the value is stored to. When loading a value from memory, we convert the dynamic signature back into a static signature that depends on the load instruction. These changes are also encoded.

Memory with Versions. The dynamic signature used for memory with versions depends additionally on the number of previously by the application executed stores (*version*). The version counter used is encoded, i.e., modifies *acc*. For a load, we have to remove the expected dynamic signature and version and replace them with the static signature of the destination register. These modifications and the signature management have to be encoded. The following listing demonstrates an ANBDmem-encoded load operation. The ANB-encoded version looks similar but does not include the version removal in line 5. The `getVersion` function returns the expected version for a given address. It is also encoded. For the implementation of this encoded version management see [18]. We use version management with check-pointing because it provides good results for applications with high and low data locality.

```

1 uint64_t loadc(ptrc, Bptr, corr){ // corr=A*Br+Bptr
2   ptr = ptrc / A;           // decode address
3   vc = load(ptr);           // load value => vc=A*r+ptr+version
4   t = (ptrc-corr)/A;         // t=((A*ptr+Bptr)-(A*Br+Bptr))/A = ptr-Br
5   rc = vc-t-getVersion(ptr); //rc=vc-(ptr-Br)-version
6   rc += (ptrc-corr) % A;     //additional check that (ptrc-corr) % A == 0
7   return rc;                 // =A*r+Br
8 }

```

The load takes an encoded pointer ptr_c , the expected signature B_{ptr} of ptr_c , and a correctional value $corr$. During encoding, we choose a value $B_r < A$ for the result's signature. Since B_{ptr} and A are also chosen at encoding time, for each call to load $corr = A * B_r + B_{ptr}$ is also constant at runtime. If a wrong or outdated address is read, the return value will not have the expected signature B_r in line 7. A store is implemented similarly.

4 Evaluation

We evaluated our approach using the following applications: `md5` calculates the md5 hash of a string, `tcas` is an open-source implementation of the traffic alert and collision avoidance system [1] which is mandatory for airplanes, `pid` is a Proportional-Integral-Derivative controller [21], `abs` implements an antilock braking system, and `primes` implements the Sieve of Eratosthenes.

Performance. Figure 1 depicts the slowdowns of encoded applications compared to their unencoded, i.e., unsafe, versions for the different codes. Time is measured for the complete application including I/O-operations.

For the AN-code, the slowdown ranges from 2 (`primes`) to 75 (`tcas`). Applications using more expensive encoded operations such as multiplications or floating point operations exhibit larger slowdowns [17]. This leads to these strongly

Fig. 1. Slowdowns of encoded application compared to their native versions.

Fig. 2. Speedup of CBE compared to SEP.

varying slowdowns. For example, `md5` contains an above average number of bitwise logical operations, which, in their encodable version, make extensive use of expensive encoded multiplications. The encoded version of `tcas` is much slower because of the extensive use of floating point operations.

The ANB-code is on average 1.9 times slower than the AN-code because it provides encoded control and data flow and the encoded operations used have to consider the signatures as well.

The slowdown of the ANBDmem-code compared to the ANB-code is on average 2.6. The main reason is the additional overhead needed to safely store and retrieve version information for dynamic memory. This overhead depends on the degree of the locality of the memory accesses executed.

One objective for CBE was to be faster than the interpreter-based SEP. Figure 2 compares for some applications the speedup of the most expensive CBE-variant (ANBDmem) compared to SEP. `tcas` and `abs` are not supported by SEP due to missing system calls. CBE always clearly outperforms SEP. We observe that the obtained speedups depend on the executed program. Especially `md5` has smaller speedups. `md5` contains an above average number of bitwise logical operations. However, SEP is incomplete. It especially does not support encoded versions of bitwise logical operations, shift operations, and casts. Those operations are just executed unencoded in SEP while they are encoded by CBE.

Error Detection. For evaluating the error detection capabilities of our encoded programs we used our error injector EIS [12]. It injects the software-level symptoms of possible hardware failures. We injected the following symptoms: exchanged operands, exchanged operators, faulty operations, modified operands, and lost stores. Further errors can be represented by combinations of these symptoms.

We applied those errors in three different modes:

Deterministic (Det) injects per run exactly one error. We execute approximately 50,000 runs for each benchmark and protection mechanism: 10,000 for each symptom. In each run another error is triggered. This tests the ability of a detection mechanism to cope with seldom occurring errors.

Probabilistic (Prob) injects an error with a probability given by us. We use the same error probability for all error detection mechanisms evaluated. At each possible point where an error (of any symptom) could be triggered an error is injected with the given probability. Thus, one execution might be hit by several different errors. With this mode we executed 6,000 runs per benchmark and per detection mechanism.

Permanent errors (Per) injects permanently faulty operation errors simulating permanent logic errors in the processor. Permanent errors are only applied to arithmetic integer operations, and loads and stores of integer values. We are injecting approximately 1,700 different permanent errors per benchmark, per detection mechanism – one error only per run.

All example applications are of similar size and we distribute the injections evenly over the program execution. Hence, with our fixed number of fault injection runs we achieve similar coverages for all applications. We chose the number of fault injection runs so that the experiments complete in a feasible time.

We compared the results of injection runs to the results of an error-free run to determine if the error injected resulted in a silent data corruption (SDC), i.e., a failure of the error detection, in an abort or a correct output, i.e., the error was masked. Figure 3 presents the results of the described error injection experiments. It focuses on the amount of SDCs because these identify a failure of the detection. Note the logarithmic scale.

We make the following observations: We see that in contrast to native, i.e., unprotected, programs the AN-encoded versions dramatically reduce the amount of SDCs, i.e., undetected errors. However, the AN-encoded versions still have a considerable amount of SDCs: on average 0.96%. The highest rate of undetected errors is 7.6% for `abs` and `det`. ANB-encoding reduces the amount of undetected errors to on average 0.07%. ANBdmem-encoding again halves the rate to on average 0.03% SDCs.

In contrast to unprotected applications, none of the encoded versions – independent of the code used – is vulnerable to permanent errors. Probabilistically (Prob) injected errors are also more often detected. The reason is that for both injection modes programs are more often hit by several errors. This increases the probability of detection as we have shown in [18].

Fig. 3. Error injection results: Amount of SDCs for all error injection runs.

To show the advantage of ANB and ANBDmem-code over AN-code we compare the overhead with the detection rate. On average, the ANB-code has a about 14 times higher error detection rate than the AN-code while the slowdown increases on average only 1.9 times. The ANBDmem-code has an about 32 times higher detection rate than the AN-code which comes at the cost of an about 5 times higher slowdown. Both the ANB- and the ANBDmem-code can compensate for their longer runtime with an disproportionately higher detection rate.

5 Related Work

Control flow checking, which can be implemented in hardware, e.g., [4], or software, e.g., [5], provides means to recognize invalid control flow for the executed program, that is, execution of sequences of instructions that are not permitted for the executed binary. In contrast to encoding, control flow checking cannot detect errors which do only influence processed data. Usually control flow checking is

only done for inter-basic-block control flow. Our ANB- and ANBDmem-encoded programs are checked on the instruction level.

Algorithm-based fault tolerance [15] and self-checking software [20] use invariants to check the validity of the generated results. Appropriate invariants have to exist which provide a good failure detection capability. They are not easy – if not impossible – to find for most applications.

Other software approaches work with replicated execution and comparison (voting) of the obtained results. The protected software is modified during or before compilation – rarely, dynamic binary instrumentation is used [11]. Replication is applied at different levels of abstraction. Some approaches duplicate single instructions within one thread, e. g., [7, 11, 4]. Others execute duplicates of the whole program using several threads, e. g., [16]. For all approaches that are based on replication, it is not possible to provide guarantees with respect to permanent hardware errors [13].

Instead of duplication, or in addition to, arithmetic codes can be used to detect errors. Therefore, the program and the processed data are modified. AN-encoding was already used by [10, 7, 14]. For all approaches the error injection experiments show a non-negligible amount of undetected failures. This is even higher for [10] and [7] because the used encoding is incomplete.

Forin’s Vital Coded Processor (*VCP*) [8] and Software Encoded Processing (*SEP*) [18] use both an ANBD-code. We compared both approaches already to CBE in Sec. 3.

6 Conclusion

We introduced compiler based encoding (CBE) – especially control flow encoding using ANB- and ANBDmem-encoding. Our experiments have shown that these two new encodings reduce the number of undetected errors more than AN-encoding. The reduction of undetected errors is higher than the increase in runtime that has to be paid for the more sophisticated protection of ANB- and ANBDmem-encoding. Thus, safety engineers can balance the error detection coverage and the performance overhead by choosing the appropriate arithmetic encoding.

Our second goal was to provide a faster encoding mechanism than SEP. We clearly achieved this goal: on average, ANBDmem-encoded applications are 108 times faster than their SEP version. Furthermore, CBE is more complete than SEP. In contrast to CBE, SEP does not support encoded bitwise logical operations, casts, shifts, and floating point operations.

References

1. The Paparazzi Project. http://paparazzi.enac.fr/wiki/Main_Page, 2009.
2. A. Avizienis. Arithmetic error codes: Cost and effectiveness studies for application in digital system design. In *Transactions on Computers*, 1971.

3. Hugh J. Barnaby. Will radiation-hardening-by-design (RHBD) work? *Nuclear and Plasma Sciences, Society News*, 2005.
4. C. Bolchini, A. Miele, M. Rebaudengo, F. Salice, D. Sciuto, L. Sterpone, and M. Violante. Software and hardware techniques for SEU detection in IP processors. *J. Electron. Test.*, 2008.
5. Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. Software-based transparent and comprehensive control-flow error detection. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, Washington, DC, USA, 2006. IEEE Computer Society.
6. Shekhar Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 2005.
7. Jonathan Chang, George A. Reis, and David I. August. Automatic instruction-level software-only recovery. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, Washington, USA, 2006.
8. P. Forin. Vital coded microprocessor principles and application for various transit systems. In *IFA-GCCT*, pages 79–84, Sept 1989.
9. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization (CGO)*, USA, 2004. IEEE Computer Society.
10. Nahmsuk Oh, Subhasish Mitra, and Edward J. McCluskey. ED4I: Error detection by diverse data and duplicated instructions. *IEEE Trans. Comput.*, 51, 2002.
11. George A. Reis, Jonathan Chang, David I. August, Robert Cohn, and Shubhendu S. Mukherjee. Configurable transient fault detection via dynamic binary translation. In *Proceedings of the 2nd Workshop on Architectural Reliability (WAR)*, 2006.
12. Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. Slice Your Bug: Debugging Error Detection Mechanisms using Error Injection Slicing. In *Eighth European Dependable Computing Conference (EDCC 2010)*, 2010.
13. Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. Software-Implemented Hardware Error Detection: Costs and Gains. In *The Third International Conference on Dependability (DEPEND 2010)*, 2010.
14. Ute Schiffel, Martin Süßkraut, and Christof Fetzer. AN-encoding compiler: Building safety-critical systems with commodity hardware. In *The 28th International Conference on Computer Safety, Reliability and Security (SafeComp 2009)*, 2009.
15. V.K. Stefanidis and K.G. Margaritis. Algorithm based fault tolerance : Review and experimental study. In *International Conference of Numerical Analysis and Applied Mathematics*, 2004.
16. Cheng Wang, Ho seop Kim, Youfeng Wu, and Victor Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *International Symposium on Code Generation and Optimization (CGO)*, 2007.
17. Ute Wappler and Christof Fetzer. Hardware failure virtualization via software encoded processing. In *5th IEEE International Conference on Industrial Informatics (INDIN 2007)*, 2007.
18. Ute Wappler and Christof Fetzer. Software encoded processing: Building dependable systems with commodity hardware. In *The 26th International Conference on Computer Safety, Reliability and Security (SafeComp 2007)*, 2007.
19. Ute Wappler and Martin Müller. Software protection mechanisms for dependable systems. *Design, Automation and Test in Europe (DATE '08)*, 2008.
20. Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *J. ACM*, 1997.
21. Tim Wescott. PID without a PhD. *Embedded Systems Programming*, 13(11), 2000.