**Abstract**—Transient and permanent errors in memory and CPUs occur with alarming frequency. Although most of these errors are masked at the hardware level or result in crashes, a non-negligible number of them leads to Silent Data Corruptions (SDCs), i.e., incorrect results of computations. Safety-critical programs require a very high level of confidence that such faults are detected and not propagated to the outside. Unfortunately, state-of-the-art fault detection techniques generally assume a limited Single Event Upset fault model, concentrating only on transient faults.

We present Δ-encoding: a software-only approach to detect hardware faults with very high probability. Δ-encoding makes no assumptions on the rate and type of faults. Our approach combines AN codes and duplicated instructions to harden programs against transient and permanent hardware errors. Our evaluation shows that Δ-encoding detects 99.997% of all injected errors with performance slowdown of 2–4 times.

### I. INTRODUCTION

A dramatic decrease in hardware reliability, most importantly in CPUs and RAM, was forecast already in the 2000s [1]. This is due to the decrease of feature sizes with each new hardware generation, causing variations in transistor behavior. These variations, if not masked at the hardware level, can lead to silent data corruptions (SDCs) in a program. Moreover, additional effects such as transistor aging and soft errors (due to alpha particles and cosmic rays hitting silicon chips) increase the probability of a program to produce wrong results.

Recent studies provide supporting evidence for this forecast. Google analyzed DRAM failure patterns across its server fleet [2]. The research concluded that (1) DRAM failure rates are higher than previously expected, (2) memory errors are strongly correlated, and (3) memory errors are dominated by hard errors rather than by soft errors. Another study shows that even ECC-enabled DRAM chips do not provide adequate protection from the emerging problem of disturbance errors, when accesses to one DRAM row corrupt data in adjacent rows [3].

Similar findings were revealed in regard to modern CPUs. Microsoft conducted analysis of hardware failures on a fleet of 950,000 machines [4]. This work showed that (1) failure rates of modern CPU subsystems are non-negligible, (2) failure rates increase with the increasing CPU speed, and (3) CPU faults tend to be intermittent rather than transient. Unfortunately, the study considers only crash failures and not data corruptions in applications; other studies, however, indicate that CPU faults result in a non-trivial number of SDCs [5].

Many hardware errors, either in CPU or in memory, lead to a process or machine crash. Still, some hardware faults induce programs to output incorrect results, which can propagate further and lead to catastrophic consequences. One anecdotal evidence is the famous Amazon S3 unavailability incident, when a single bit corruption in a few messages caused an 8-hour outage [6].

The consequences are even more disastrous in safety-critical applications. As one example, Toyota Motor Corporation was forced to recall its automobiles in the years 2009–2011 after several reports that Toyota cars experienced unintended acceleration [7]. The number of victims was estimated to be 37, and financial loss for Toyota $2.470 million. Though the exact causes of the problem were not found out, insufficient protection against hardware errors could be one of them:

> “Michael Barr of the Burr Group testified that ...Toyota did not follow best practices for real-time life-critical software, and that a single bit flip which can be caused by cosmic rays could cause unintended acceleration.”

Detecting hardware faults of all types is a necessity for applications from different domains. Tolerating faults once they are found can often be achieved by simply restarting the process or rebooting the machine; in most cases it is enough that incorrect computation results are not propagated to the outside. Therefore, we concentrate on hardware error detection in this paper.

The conservative error detection approach, widely used in automotive and aerospace systems, is to employ some form of hardware-based fault tolerance. Usual mechanisms include triple/dual modular redundancy (TMR/DMR), flip-flop hardening, watchdogs, etc. [8]. The hardware-based approach, however, implies higher hardware costs and lower performance in comparison to modern commodity hardware. For example, Intel conjectures that future self-driving cars will require greater computing power and suggests to use commodity CPUs [9].

Another approach called Software-Implemented Hardware Fault Tolerance (SIHFT) [10] achieves fault tolerance via software-only methods; thus, it does not require specialized

---

1One third of machines under study experienced at least one correctable memory error per year; the annual rate of uncorrectable errors amounted to a significant 1.3%. Note that all memory modules were equipped with error correcting codes (ECC).

2For example, the chance of a crash is 1 in 190 for machines with the total CPU time of 30 days.

3More precisely, we concentrate on detection of data corruptions that occur due to hardware errors changing a program’s data flow.
hardware. However, in spite of the experimental studies clearly indicating the prevalence of permanent and intermittent errors in CPU and memory, most SIHFT techniques assume only transient errors. In this sense, these techniques favor performance over fault coverage and cannot be relied upon in safety-critical systems.

One notable SIHFT technique that can detect both permanent and transient errors in underlying hardware is encoded processing [11]. It is based on the theory of arithmetic codes (AN-encoding) and was used in fault-tolerant computing [12]. Unfortunately, pure AN-encoding has limited fault coverage. Advanced variants of AN-encoding exist [11], but programs encoded with them – namely with the ANBD variant – experience slowdowns of up to 250x. Thus, though ANBD-encoding yields very high fault coverage, it is impractical in terms of performance.

As a result, existing SIHFT techniques either do not detect all possible hardware errors or incur prohibitive performance penalties. This work makes a step towards hardening critical computations against permanent and transient hardware errors with a moderate performance penalty.

Our approach, called \( \Delta \)-encoding, is based on the combination of AN-encoding and duplicate execution of instructions. The original program data flow is duplicated and AN-encoded at compile-time; at run-time, the program effectively works on two copies of data encoded in two different ways. The careful choice of AN-encoding parameters coupled with execution duplication greatly simplifies AN-encoded operations, improving the performance; moreover, the combination of approaches facilitates detection of all types of hardware errors.

We implemented \( \Delta \)-encoding as a source-to-source transformer. Our fault injection experiments reveal that \( \Delta \)-encoding can detect, on average, 99.997% of injected errors. Our performance evaluation shows that \( \Delta \)-encoding incurs an acceptable slowdown of 2–4 times in comparison to native execution.

II. BACKGROUND

The \( \Delta \)-encoding technique proposed in this paper combines two existing approaches: AN-encoding and duplicated instructions. In this section, we briefly discuss both of them.

A. AN-encoding

AN-encoding is a technique to protect program execution from transient and permanent errors in the underlying hardware. It is based on AN codes – error correcting codes suitable for arithmetic operations [13]. Schiffel [11] describes AN-encoding and its variants in detail.

With AN codes, to encode an integer \( n \), one multiplies it by a constant \( A \). The resultant integer \( \hat{n} = A \cdot n \) is called a code word; all words that are not multiples of \( A \) are invalid. If a hardware error alters \( \hat{n} \), it becomes an invalid word with high probability; this dependence on \( A \) [11]. If \( \hat{n} \) is still a code word, \( \hat{n} \mod A = 0 \); if the result of this operation is not 0, a hardware error is detected. To decode, a division \( \hat{n}/A \) is used.

AN-encoding exploits information redundancy, i.e., additional bits are required to store an encoded integer. In practice, the number of bits to represent encoded integers is doubled.

As an example, consider the addition of two integers 5 and 3 (see Listing 1). For simplicity, we choose \( A = 11 \). AN-encoded integers are thus \( A \cdot 5 = 55 \) and \( A \cdot 3 = 33 \). These code words can be directly added and result in a code word: \( 55 + 33 = 88 \). Now, if a hardware error would cause any of the terms to become invalid, the sum will also be an invalid code. Listing 2 shows an AN-encoded version of the original addition.

This example highlights two main properties of AN-encoding: first, operations on encoded inputs directly produce encoded outputs; second, errors in inputs propagate to outputs. The first property means that by substituting all original operations with encoded operations, the data flow of a program is protected against hardware faults. The second property implies that the encoded execution of a program does not require intermediate checks.

One drawback of AN-encoding is that not all operations on encoded values are easily implemented. As the previous example shows, encoded addition corresponds to the usual arithmetic addition; subtractions and comparisons are also trivial. However, encoded multiplication, division, bitwise operations, etc. require more sophisticated implementations. These complex encoded operations can hamper performance and/or require intermediate decoding of operands.

Another drawback of pure AN-encoding is that it does not detect all types of hardware errors. In our previous example, if

<table>
<thead>
<tr>
<th>Listing 1: Native program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 int32_t a = 5;</td>
</tr>
<tr>
<td>2 int32_t b = 3;</td>
</tr>
<tr>
<td>3 int32_t c = a + b;</td>
</tr>
<tr>
<td>4 printf(&quot;%d&quot;, c);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 2: AN-encoded program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 #define A 11</td>
</tr>
<tr>
<td>2 int64_t a = 5 * A;</td>
</tr>
<tr>
<td>3 int64_t b = 3 * A;</td>
</tr>
<tr>
<td>4 if ((c % A) != 0) raise_error();</td>
</tr>
<tr>
<td>5 printf(&quot;%d&quot;, c/A);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 3: Duplicated instructions program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 int32_t a1 = 5;</td>
</tr>
<tr>
<td>2 int32_t a2 = 5;</td>
</tr>
<tr>
<td>3 int32_t b1 = 3;</td>
</tr>
<tr>
<td>4 if ((c1 != c2) raise_error();</td>
</tr>
<tr>
<td>5 printf(&quot;%d&quot;, c1);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 4: ( \Delta )-encoded program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 #define A1 9</td>
</tr>
<tr>
<td>2 int64_t a1 = 5 * A1;</td>
</tr>
<tr>
<td>3 int64_t b1 = 3 * A1;</td>
</tr>
<tr>
<td>4 if ((c1/A1 != 0)</td>
</tr>
<tr>
<td>5 printf(&quot;%d&quot;, (c1 - c2) &gt;&gt; 1);</td>
</tr>
</tbody>
</table>
the addition operation is erroneously substituted by subtraction, the result is still a code word, since $55 - 33 = 22$. Moreover, if one of the operands is replaced by some other code word (due to a fault on the address bus), the result is also a code word, e.g., $55 + 11 = 66$. To detect these types of errors, variants of AN-encoding were developed, namely ANB- and ANBD-encodings [14]. Unfortunately, they incur very high performance penalties (up to $250x$) rendering them impractical in most use cases.

AN codes should not be confused with conventional linear codes such as Hamming codes or Reed-Solomon codes. Firstly, the linearity property does not hold in AN codes; secondly, linear codes are suitable for storage and transmission whereas AN codes are used in data processing.

In general, AN-encoding has the advantage of detecting both transient and permanent errors during program execution; a severe disadvantage is its low performance. Pure AN-encoding cannot detect all kinds of hardware errors and thus it does not provide high fault coverage. ANB- and ANBD-encodings do provide full fault coverage, but at the price of even higher performance overheads.

B. Duplicated Instructions

Fault tolerance can also be achieved by duplicating all original instructions in a program. The duplicates work with a second set of registers and variables, i.e., all data is also duplicated. During execution, “master” and “shadow” instructions are issued on the same processor; their results are compared periodically to check for hardware errors. Oh, Shirvani and McCluskey [15] provide detailed information about error detection by duplicated instructions.

Concerning our previous example of $5 + 3$, the addition operation is issued twice on the CPU, such that two copies use two different sets of registers. The check operation makes sure that both copies calculated 8, and if not, a hardware error is detected. Listing 3 illustrates this.

The duplicated instructions approach assumes that hardware faults are transient and affect only one data-flow copy. For example, this approach cannot detect hard errors in the CPU. If the addition operation is permanently faulty, then $5 + 3$ can result in an incorrect value for both copies.

The duplicated instructions technique incurs only modest performance penalty of 60% [16], since additional instructions can be effectively scheduled by the compiler and executed by the CPU in an out-of-order fashion. Indeed, since “master” and “shadow” execution paths are independent of each other and require synchronization only at rare check points, the execution runs essentially in parallel on modern super-scalar processors.

On the whole, the approach of duplicated instructions enables comprehensive protection from transient errors, incurring only modest execution slowdowns. However, this approach cannot cope with permanent errors affecting both “master” and “shadow” copies of data flow.

III. Fault Model

We adopt a data-flow software-level symptom-based fault model from [11]. This model provides an abstraction of the underlying hardware and works on the “symptoms” caused by hardware errors at the software level. Such a model has several advantages: (1) it is independent from specific hardware models and thus applies to any combination of CPU/RAM, (2) it does not account for masked hardware faults, i.e., faults that are neutralized at hardware level, and (3) this fault model can be easily adapted for fault injection campaigns.

The model consists of the following symptoms:

**Modified operand** One operand is modified, e.g., $55 + 33$ is changed to $51 + 33$. This happens due to a bit flip in memory/CPU register.

**Exchanged operand** A different but valid operand is used, e.g., $55 + 33$ is changed to $55 + 11$. This happens due to a fault on the address bus.

**Faulty operation** An operation produces incorrect results on specific inputs, e.g., $55 + 33$ results in 87. A CPU design flaw can lead to such a fault.

**Lost update** A store operation is omitted, e.g., the result of $55 + 33$ is not stored in memory/CPU register; an outdated value from the memory/register is then erroneously used. This happens due to a fault on the address bus.

Many fault-detection approaches assume a Single Event Upset (SEU) fault model, where exactly one bit is flipped throughout program execution; in contrast, we make no assumptions on the number of bits affected by a hardware error or on the number of hardware errors during execution.

We argue that the SEU model is unrealistic. First, as studies show ([2], [3]), modern RAM experiences not only transient bit flips, but also permanent faults. Second, another study [17] reveals that about 17% of hardware faults affecting the combination logic result in double or multiple bit errors. These results motivate the adoption of a fault model that has no error rate assumption: any number of errors of any type can happen during program execution. Our only assumptions are that errors occur randomly and corrupt a random number of bits.

Our fault model does not cover control flow errors, when a corrupted instruction pointer (IP) points to an unintended instruction address. Such faults have a very low probability of resulting in SDC. Nevertheless, our approach can be coupled with a control flow checker to detect both data and control flow errors.

Finally, the sphere of replication (SoR) [18] assumed in this paper is the CPU and the memory directly used by the encoded program (or the encoded part of a program). The operating system as well as the disk and network subsystems are out of SoR; errors in these systems cannot be detected.

IV. $\Delta$-Encoding

In this section, we describe $\Delta$-encoding, a novel technique that combines AN-encoding and duplicated instructions. $\Delta$-encoding borrows the ability to detect hard errors from AN-encoding; it uses the idea of duplicated instructions to achieve full fault coverage without sacrificing performance. Moreover, a clever combination of approaches allows to simplify AN-encoding, improving its performance.
Conceptually, Δ-encoding performs two compile-time transformations on the original program: first, all data is AN-encoded and all original operations are substituted by AN-encoded operations, second, all encoded data and operations are duplicated and checks are inserted at synchronization points. The result is a hardened program with two copies of a completely encoded data flow, as shown in Fig. 1.

A. Encoded Data

To encode data in Δ-encoding, we set two different constants for the two copies of data: $A_1$ for the first encoded copy and $A_2$ for the second copy. Thus, the two copies of data flow operate on different values, i.e., our approach employs data diversity, which is beneficial for fault tolerance [19]. In particular, if a hard CPU fault triggers on some specific inputs, it will corrupt only one copy of the data, but not the other.

The key idea behind Δ-encoding is a smart choice of $A_1$ and $A_2$:

$$A_1 - A_2 = 1 \quad \text{(1)}$$

This choice of the constants enables us to decode values quickly, by subtracting the second encoded copy $\hat{n}_2$ from the first encoded copy $\hat{n}_1$ (hence the name Δ-encoding):

$$n = \hat{n}_1 - \hat{n}_2 = n \cdot A_1 - n \cdot A_2 = n \cdot (A_1 - A_2) \quad \text{(2)}$$

Note that this decoding requires only one instruction cycle; in contrast, decoding in pure AN-encoding is much more expensive, since it requires a division instruction4. Such quick decoding is especially beneficial for programs that have heavy use of pointers because all pointers are kept encoded and must be decoded at each pointer dereference.

The choice of $A_1$ and $A_2$ in Equation 1 has a drawback: both copies of a value are decoded in the same way (by subtracting the $A_2$-encoded copy from the $A_1$-encoded copy). This can lead to SDC since a permanent fault affects both decoding operations in the same way. Thus, we push the idea further and use the following scheme to choose $A_1$ and $A_2$:

$$A_1 = 2^k + 2^i \quad A_2 = 2^k - 2^i \quad \text{(3)}$$

where $k$ and $i$ are non-negative integers, $k > i$.

We notice that:

$$A_1 - A_2 = 2^k + 2^i - 2^k - 2^i = 2^{i+1} \quad \text{(4)}$$

$$A_1 + A_2 = 2^k + 2^i + 2^k - 2^i = 2^{k+1} \quad \text{(5)}$$

Based on Equations 4 and 5, there are two ways to decode a value:

$$n = (\hat{n}_1 - \hat{n}_2)/2^{i+1} = n \cdot (A_1 - A_2)/2^{i+1} \quad \text{(6)}$$

$$n = (\hat{n}_1 + \hat{n}_2)/2^{k+1} = n \cdot (A_1 + A_2)/2^{k+1} \quad \text{(7)}$$

The division by a power of 2 corresponds to the right shift instruction. Since we fix $k$ and $i$ beforehand, the number of bits to shift by is known at encoding time. As a result, decoding schemes 6 and 7 require only two cycles: one for subtraction/addition and one for right shift.

For example, let $k = 3$ and $i = 0$. Then $A_1 = 9$ and $A_2 = 7$; their difference is $A_1 - A_2 = 2$ and their sum is $A_1 + A_2 = 16$, and to decode one needs to shift right by $i + 1 = 1$ and $k + 1 = 4$ correspondingly. Our original code snippet from Listing 1 can be Δ-encoded with these parameters and results in an encoded program from Listing 4.

Δ-encoding uses this scheme, with $A_1$ and $A_2$ chosen as in Equation 3 and decoding as in Equations 6 and 7. This scheme has two advantages: (1) decoding is much faster than in pure AN codes and (2) two different ways to decode a value will fail differently in reaction to the same permanent error.

We introduce these parameters here for clarity of description; the justification for the parameters is given in Section V-A.

B. Encoded Operations

Δ-encoding works on AN-encoded values. This implies that all original operations – addition, subtraction, multiplication, bitwise AND, OR, XOR, shifts, comparisons, etc. – are substituted with the corresponding encoded operations. In this section, we provide examples of some typical Δ-encoded operations. For clarity, we introduce them as functions in the C language.

Encoding and decoding operations were already described conceptually. Listings 5 and 6 show their practical implementations. It is worth mentioning that encoding could be implemented through shifts and addition/subtraction, as shown by Equation 3; however a simple multiplication exhibits similar performance. The decoding operation corresponds to Equations 6 and 7.

4The division instruction is one of the most costly operations in modern CPUs. For example, according to the Intel IA-64 architecture manual, division takes 60-80 cycles to finish [20].
Some operations require partial decoding. One example is a left shift operation: the number of bits by which an integer itself can stay encoded (see Listing 8). Another example is multiplication, where it is enough to decode only one operand.

Finally, bitwise operations (AND, OR, XOR, one’s complement) as well as division are notoriously slow if implemented using encoding. In these cases, the only reasonable strategy is to decode operands, perform the original operation, and re-encode the result. Listing 9 exemplifies this using the XOR operation.

Encoded operations must be not only fast, they must also propagate possible errors to the resultant integer. This holds for operations like addition. Operations like left shift and XOR rely on duplicated instructions, since it is unlikely that the result of the first operation execution (with \( A_1 \)) will be corrupted exactly in the same way as in the second execution (with \( A_2 \)). Moreover, the sum of two encoded copies \( x_1 + x_2 \) has zeros in the lower 14 bits by Equation 5 (otherwise it indicates that an error occurred during the operation); we use this property to propagate errors in some operations.

C. Accumulation of Checks

As any fault detection mechanism, \( \Delta \)-encoding inserts periodic checks of calculated values. An example of such a check is shown in Listing 4, Line 5. It includes checking if both copies of a variable are code words and if they correspond to the same original value. If any of the conditions fails, then an error must have happened, and the execution is terminated.

A naive approach to detect errors would be to check the result of each encoded operation. This would lead to a tremendous slowdown, since each operation would then be accompanied by a heavy-weight check with divisions and branches.

On the other side, one could check only final results, i.e., check only output values right before decoding them. Indeed, if the property of error propagation would hold for all encoded operations, it would be sufficient to check only the results of the computation. In real-world scenarios, however, this property is frequently violated; the XOR operation from Listing 9 is one example.

The practical solution would be to analyze the program’s data flow and insert checks only at critical points (e.g., after each XOR operation, but not after additions). Even in this case, the number of inserted checks incurs significant overhead.

To achieve a better trade-off between performance and fault coverage, we introduce the accumulation of checks. We allocate a pair of integers called accumulators and we substitute all intermediate checks with a simple addition to the accumulators. The principle is illustrated in Listing 10. The original program performs two operations: addition \( x + y \) and subtraction \( x - y \). The encoded program adds two accumulations and one subsequent check instead of two expensive checks.

Using accumulators instead of direct checks is beneficial for performance: accumulation requires only two additions instead of several divisions and branches. Moreover, it does not decrease the error detection capabilities of \( \Delta \)-encoding, because the addition operation propagates any erroneous value to the accumulator. One last non-obvious advantage is that accumulations are less susceptible to the “who guards the guardians?” problem: a check could be erroneously skipped due to a single CPU fault, but quietly skipping both accumulator updates is highly improbable.
D. Fault Coverage

Δ-encoding provides very high fault coverage. Here we explain how our approach covers all symptoms from the symptom-based fault model described in Section III. We provide a quantitative analysis only for the case of modified operand due to lack of space; other symptoms can be analyzed in a similar way.

**Modified operand** AN codes guarantee that, given a modified operand fault, the probability of a SDC is \(1/A\) ([11]). In duplicated instructions, given that a random fault (corrupting a random number of bits) affected both copies of the operand, the probability of a SDC is \(1/2^n\), where \(n\) is the number of bits of the operand.

With Δ-encoding, given that a fault affected both copies of the operand, a may happen only if (1) the first copy is a code word and (2) the second copy corresponds to the first copy (i.e., produces the same original value when decoded). Combining these requirements together and taking into account that AN codes double the number of bits in operands, we get the probability of a SDC equal to \(1/A \cdot 2^{2n}\).

**Exchanged operand** Since Δ-encoding performs each operation twice, SDC happens only if two exchanged operand faults substitute two correct copies with two incorrect but valid copies. The probability of such chain of events is negligible.

**Faulty operation** Two copies of data are encoded differently (with \(A1\) and \(A2\)) in Δ-encoding; thus, two executions of a faulty operation would work on different operands and would fail in different ways. This means that the probability that two faulty operations produce two corresponding code words is negligible.

**Exchanged operation** Since Δ-encoding performs each operation twice, two copies of the operation must be substituted by two exactly the same non-intended operations. This scenario is highly improbable.

**Lost update** In Δ-encoding, two store operations are used to update two copies of data; thus, two stores must be omitted to result in a lost update. Such scenario has negligible probability.

As this analysis shows, Δ-encoding provides high fault coverage for all types of faults. Notice that modified operand faults happen more frequently than other types, because the underlying hardware errors – memory and CPU register bit-flips – occur with perceptible regularity. But other types of faults, however improbable they are, must also be accounted for in safety-critical systems.

The combination of duplicated instructions, AN codes and heuristic accumulation of checks also provides high guarantees against intermittent and permanent errors. For example, using duplicated instructions alone, it is possible that both copies of a variable are stored in the same physical CPU register which experiences a stuck-at fault, and thus the fault remains undetected. In Δ-encoding, a stuck-at fault in a register results in an invalid word (with high probability).

Interestingly, the approach of duplicated instructions cannot detect permanently faulty operations. If the same inputs are fed to two executions of a faulty operation, both executions produce the same incorrect output. In Δ-encoding, the two copies of data are diverse, leading to two different incorrect results. Thus, Δ-encoding can detect permanent faults which would lead to a SDC in the case of simple duplicate execution.

V. Implementation

We implemented Δ-encoding as a source-to-source C transformer in Python (see Fig. 2). Original C programs are encoded at the level of an Abstract Syntax Tree (AST) built by PycParser. Our transformer walks through the AST, substituting all inputs and constants by encoded values and all original C operators by the corresponding encoded operations. The transformer also produces function-wrappers to perform libc/system calls from encoded source (e.g., malloc()) and vice versa.

Δ-encoded programs preserve the original code structure, i.e., original control flow as well as variable and function names. This is possible because our transformer does not employ any code optimizations, working as close to the original source as possible. Preserving the original information greatly facilitates debugging and manual changes in encoded programs.

The Δ-encoded code emitted by the transformer does not rely on a specific compiler and is not influenced by compiler optimizations. The structure of Δ-encoding itself prevents the compiler from optimizing duplicate instructions away. As an example, consider the decoding operation from Listing 6: the compiler has no knowledge of inherent interdependency between two encoded copies and cannot figure out that the two ways of decoding produce the same original value.

The Δ-encoded code can be intermingled with unencoded sources. First, the programmer can manually add calls to unencoded functions in the emitted encoded code (e.g., adding printf() calls for debug purposes). Second, the transformer generates wrappers for unencoded functions used by the encoded code (e.g., libc functions such as malloc() and free()).

A. Encoding Data

Since Δ-encoding expands the original domain of values to accommodate all encoded values, our implementation restricts all integer variables to be at most 48 bits wide. We chose \(A1 = 8193\), \(A2 = 8191\) such that the encoded values never exceed the 64-bit range, since the maximum encoded value \((2^{48} - 1) \cdot 8193\) is less than 64 bits wide.

\(^5\)https://github.com/eliben/pycparser

\(^6\)Compiler optimizations are a constant threat for fault-tolerant high-level transformations, since they can be very efficient at eliminating code and data redundancy. Some techniques even require all compiler optimizations to be disabled, as in [21].
In general, original integer types are limited by at most 32-bit data types. 64-bit types are also supported, but the original program must guarantee that the values never exceed the 48-bit bound. This is the case for pointer types: on modern 64-bit systems, pointers are 64 bits wide but virtual address formats use only the 48 low-order bits [22]. Therefore, our implementation supports pointer types on current 64-bit architectures.

The $\Delta$-encoding transformer implements two copies of variables as two-item arrays. For example, $\text{int32}_T n$ is transformed into $\text{int64}_T n\_{\text{enc}}[2]$. This implementation is not optimal with respect to fault detection, because the two copies of the variable are adjacent to each other, and one fault changing bits in-between can corrupt both copies. A better implementation would require separate “shadow” stack and heap for second copies of data. Unfortunately, such separation would require compiler support and thus is impossible in our current C-to-C transformer approach; we leave it as future work.

One interesting feature of $\Delta$-encoding is the prohibition of silent integer over- and underflow. AN codes modulo arithmetic is not isomorphic to the original modulo arithmetic, e.g., $2^{32} \cdot A$ would not wrap to 0; $\Delta$-encoding would therefore require expensive checks to support integer overflow behavior. Wishing to keep $\Delta$-encoded programs as fast as possible, we disallow all silent under- and overflows. If a programmer wishes to support such wraparounds, she is required to implement them explicitly. Our decision is also partially justified by security reasons: many integer overflows are unintended and can be a source of vulnerabilities [23]. In $\Delta$-encoding, silent integer over- and underflows raise a run-time error.

There is one subtle issue when encoding local loop variables. Modern compilers are particularly good at optimizing loops; in several occasions we noticed that the compiler removed the second copy of a loop variable, weakening the protection. Indeed, the compiler has full right to perform such an aggressive optimization: it knows an initial value and the complete data flow of a loop variable and ascertains uselessness of the second copy. To prevent the compiler from removing the variable, we insert inline pseudo-assembly that clobbers both copies of the loop variable. This example illustrates how careful one should be when enabling fault tolerance without changing the compiler behavior.

B. Encoding Operations

Some of the encoded operations were already described in Section IV-B. The final implementations follow closely the examples from Listings 5 to 9. The $\Delta$-encoding transformer provides the complete set of encoded C operators, including arithmetic, comparison, logical, bitwise, member and pointer operators, casts, etc.

All encoded operations are inlined in the final executable. This enables the compiler to choose the specific computation path. For example, the decode() operation from Listing 6 will be inlined two times in the code (first with $A1$ and then with $A2$): first time stripped to Line 3, second time – to Line 5.

The code emitted by the $\Delta$-encoding transformer must be compiled with the SSE extensions disabled. Otherwise the compiler can glue two move-to-memory instructions of adjacent data copies into one SSE-move. If a hardware error affects this SSE-move, both copies of data are affected, which can lead to undetected SDCs. This flaw in our data representation, where variables are encoded as two-item arrays, was already described in the previous subsection. Note that if copies of data would be completely decoupled, SSE extensions could be enabled again.

The AN codes approach is not able to detect incorrect branching resulting from faults in branching logic. Indeed, the decision of which branch to take is based on the flag bit values of a status register. Flag bits cannot be encoded, and a single bit-flip can lead to an incorrect branch. Fortunately, duplicated instructions suggest a way to detect errors in branching logic: our transformer inserts a “shadow” branch for each original branch. The original branch is encoded to work on the first copy of data, the “shadow” branch works on the second copy. If the branching decisions differ in the two branches, an error is detected.

C. Accumulation of Checks

The idea of accumulation was defined in section IV-C; here we describe some implementation issues.

As explained previously, accumulations are a low-overhead substitute for checks, such that the frequency of the checks themselves is significantly decreased. In fact, our experiments showed that checks can be done in the very end of computation, and all intermediate steps are sufficiently protected via accumulations. In the final implementation, we introduced checks only at the end of encoded computations and in wrapper functions.

In its turn, the frequency of accumulations can be tuned. Ideally, data flow analysis must be done to pinpoint critical places. Currently, we adopt a simple strategy: accumulations are inserted after each assignment in original C code. This straightforward technique yields satisfactory results. We leave comprehensive data flow analysis as future work.

As shown in Listing 10, accumulation corresponds to one addition operation. Accumulators are 128-bit integers\footnote{We use $\text{int128}_T$ data type provided by gcc. Under the hood, this data type is treated as two 64-bit integers.}. It is tempting to use 64-bit accumulators, but they overflow fast; the accumulation operation would require an additional overflow check. We opted for 128-bit accumulators instead. Since encoded values can be maximum 64 bits wide, $2^{64}$ accumulations must happen before the accumulators overflow in the worst case. This number of accumulations is enough for any conceivable program; overflow checks are not required for 128-bit accumulators.

Unfortunately, signed 128-bit addition is much slower than its 64-bit counterpart on modern CPUs. It requires one sign extension, one 64-bit addition and one 64-bit add-with-carry – 3 operations in total. Our performance evaluation highlights this slowdown.

Interestingly, it can be meaningful to remove all accumulations completely and perform only one check in the very end of the encoded computation. Remember that $\Delta$-encoding (ideally)
propagates all hardware errors to outputs. One can rely on this property and get rid of all intermediate accumulations, in the hope that any error will be detected by the final check. Our evaluation shows that such a trade-off between performance and fault tolerance is acceptable in some scenarios.

Moreover, accumulations and checks could be done in parallel to the program’s execution. The program could send encoded values for accumulation/check asynchronously and immediately continue execution. Accumulation/check functionality could run on another CPU core or in the dedicated hardware module. If the system allows certain latency between the actual corruption of data and its detection, this parallel approach could be used. For example, automotive embedded systems allow for such latency and are usually equipped with a special hardware watchdog⁸; it would be reasonable to add the accumulation/check functionality in the watchdog and run the encoded program on the main CPU.

VI. EVALUATION

In this section, we evaluate a set of programs encoded with the ∆-encoding transformer in terms of performance and fault coverage. The set of programs under test consists of several microbenchmarks and two use cases. Microbenchmarks give an estimation of the provided fault coverage versus performance slowdown. The first use case is taken from the field of distributed systems and exemplifies the so-called trusted modules – small safety-critical parts of applications which need to be robust against hardware errors. The second use case comes from the field of automotive embedded systems and exemplifies X-by-wire systems, where a program processes data from sensors and controls actuators such as car brakes.

A. Methodology

1) Performance Experiments: All performance experiments were run on a computer with Intel Core i5-3470 CPU (Ivy Bridge architecture), 4GB RAM, L1, L2 and L3 caches of 32KB, 256KB and 6MB. All programs and their variants (including native) were compiled using gcc version 4.8.2, with all optimizations enabled except for SSE (flags -O3 -mno-sse). For all programs, execution time was calculated as the number of cycles to perform the processing of data. All programs were run for at least one second, with predefined inputs. The final results are an average of 5 runs. All performance figures show a slowdown compared to the native execution.

Each ∆-encoded program was tested in 3 variants: with 128-bit accumulation, without accumulation, and with parallel accumulation (simulation). These variants were described in Section V-C. The variant with 128-bit accumulation (∆-full) provides full-fledged protection from hardware errors. The variant with no accumulation (∆-stripped) reduces fault coverage and increases performance, and can be an appropriate trade-off for some scenarios. Finally, “parallel accumulation” (∆-parallel) is a simulation of hardware-implemented accumulation; we simulate it by moving encoded values to a predefined memory address instead of adding them to the accumulator.

2) Fault Injection Experiments: For fault injection campaigns, we used Intel Pin⁹ and the BFI plug-in¹⁰. BFI is able to inject random faults and was used in other research [25]. We improved BFI to also inject stuck-at intermittent/permanent faults.

BFI injects single- and multiple-bit faults in: CPU register file, memory cells, address bus, and code segment. These hardware faults trigger software-level symptoms of our fault model. Modified operands are caused by bit-flips in registers and memory. Exchanged operands are due to faults on the address bus or in registers holding addresses. A faulty operation is represented as a fault in operation’s output register/memory cell. Exchanged operations are transient faults in the code segment. A lost update is a direct consequence of address corruption during a move instruction.

We conduct three fault injection campaigns:

Transient Faults A single multiple-bit transient fault is injected per run, with 100, 000 runs in total. This is similar to the Single Event Upset model, but the fault can corrupt multiple adjacent bits.

Intermittent Faults The same stuck-at fault is triggered for the duration of 100 instructions, with 20, 000 runs in total. For example, it simulates an intermittent stuck-at-1 fault in a RAX register.

Permanent Faults The same stuck-at fault is triggered for the whole duration of the computation, with 20,000 runs in total. For example, it simulates a permanent stuck-at-1 fault in a RAX register.

We inject hardware errors at random and uniformly distributed. In the case of intermittent faults, the fault is injected at a random instruction and reoccurs in 100 subsequent instructions. In the case of permanent faults, the fault is injected at a random instruction and reoccurs until the computation is finished.

The results of fault injections are sorted in 5 categories: masked faults (do not affect execution), OS-detected (detected by OS, e.g., segmentation fault), hang (the program hanged because of the fault), ∆-detected (detected by ∆-encoding), SDC (undetected; led to silent corruption of data).

Each ∆-encoded program was tested in 2 variants: with accumulation (∆-full) and without it (∆-stripped).

B. Microbenchmarks

As a proof of concept, we chose several microbenchmarks: bubblesort, quicksort, linked list, CRC32, dijkstra, and patricia trie. CRC32, Dijkstra, and patricia trie are taken from MiBench [26]. These three benchmarks perform a significant number of I/O operations to read inputs; in contrast, bubble-sort, quicksort and linked list work purely on memory values.

The performance results of the benchmarks are shown on Fig. 3 (first six). ∆-full versions incur the overhead of 4.08x on average, ∆-stripped and ∆-parallel – 2.26 and 2.59x correspondingly. ∆-parallel performs two times better than

⁸The automotive E-Gas Monitoring Concept has 3 levels of design, with the third, “controller monitoring” level implemented as an independent hardware module (watchdog) [24].

⁹http://www.intel.com/software/pintool

¹⁰https://bitbucket.org/db7/bfi
Δ-full on some benchmarks, which indicates that a hardware-assisted approach of Δ-parallel could bring a significant performance improvement.

As for fault coverage, we performed fault injection experiments on one representative benchmark – bubblesort. The results are depicted in Figs. 4–6. The native program experiences a significant number of SDCs (from 31% for permanents up to 68% for transients). Δ-encoding variants drastically reduce the rate of SDCs to almost 0%.

It is interesting to examine the few SDCs not detected by Δ-encoding. In the case of transient faults (Fig. 4), all 8 undetected faults happened on the address bus such that the injected corrupted bits were written in-between two copies of data, corrupting them both in the same way. This issue was discussed in section V-A and is a deficiency of our implementation.

In the case of intermittent and permanent faults (Figs. 5 and 6), all 6 SDCs resulted from the same corrupted register. This register was allocated by the compiler for the same encoded operation on two copies of data, such that two copies were affected by the same permanent fault. This is yet another disservice of a compiler (the first one was discussed in section V-A); these faults could be detected if we would have control over the compiler’s backend.

C. Use Case: Trusted Modules

The first use case, the HardCore trusted module, comes from the field of dependable distributed systems. HardCore is a small safety-critical part of a bigger system – HardPaxos [25]. HardPaxos is a version of the Paxos consensus protocol which enables the service on top to tolerate hardware errors; fault tolerance of the whole service depends solely on HardCore. That is, HardCore is required to have very high fault coverage.

We encoded HardCore using Δ-encoding and reproduced the experiments from [25]: for the leader and for the follower scenarios. Note that the version of HardCore described in [25] was hardened with a variant of AN-encoding called 2AN, incurring very high overheads compared to native execution. The performance numbers for 2AN and our Δ-encoding are shown in Table I; the slowdown for the worst case-scenario (for leader) is presented in Fig. 3. HardCore’s slowdown is higher in comparison to microbenchmarks: HardCore makes heavy use of small loops which the compiler unrolls for the native version but not for Δ-encoded versions (see V-A). In general, our evaluation shows that the Δ-encoded HardCore is one order of magnitude faster than the 2AN-encoded version.

The results of fault injections can be seen in Figs. 4–6. The native version has a significant number of SDCs (31% in case of transients, 11% in case of permanents), while the Δ-encoded HardCore detects all injected errors in all experiments. Note that Δ-stripped performs no worse than Δ-full: the reason is the small size of HardCore functions, such that the injected error propagates directly to the outputs. This means that the Δ-stripped version provides complete fault coverage with the average performance benefit of 70% compared to the Δ-full encoding.

D. Use Case: Safety-Critical Embedded Systems

Our second use case, which we refer to as industrial, is a real-world X-by-wire controller from the automotive embedded systems domain. The program makes heavy use of arithmetic operations, working on a small set of variables and spanning over 900 lines of code. We consider this program a typical example of safety-critical embedded applications which can benefit from Δ-encoding.

The performance slowdown is shown in Fig. 3. We would like to stress the slowdown of 4.7 times for the Δ-parallel variant: parallel accumulation on a separate hardware module is well-suited for embedded systems, since this functionality can be put in the already existing hardware watchdog. The relatively high slowdown is due to division operations, which...
Figs. 4–6 show the fault injection results for the industrial program. The Δ-full variant shows very high fault coverage, with 3 SDCs in the case of transients and 0 in other cases. The Δ-stripped variant, however, results in a significant number of SDCs: the industrial program has a long and complex execution path such that errors do not propagate to the outputs. This is in contrast to HardCore where Δ-stripped had the same fault coverage as Δ-full. The reasons for SDCs are the same as for bubblesort and HardCore.

E. Discussion

Δ-encoding was developed to provide a high level of fault tolerance. In this sense, we favor fault coverage over performance. Δ-encoded programs must be protected from all error types: transient, intermittent and permanent, single-bit and multiple-bit, single faults and multiple faults. Our experiments show that Δ-encoding (namely the Δ-full variant) achieves an average fault coverage of 99.997%.

We consider performance slowdowns of 3-4 times acceptable for our use cases. First of all, safety-critical computations are usually limited in size and not resource-demanding. Second, a software-only encoded processing approach is inherently slow, and a slowdown of several times is a significant improvement compared to the previous works on AN-encoding.

<table>
<thead>
<tr>
<th>DI</th>
<th>Δ-stripped</th>
<th>Δ-parallel</th>
<th>Δ-full</th>
<th>ANBD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.6</td>
<td>2.1</td>
<td>2.4</td>
<td>4.4</td>
<td>16.0</td>
</tr>
</tbody>
</table>

require decoding to the original values, their division and subsequent encoding (see Section IV-B).

TABLE II: Quicksort’s slowdown: comparison of approaches.

Figs. 4–6 show the fault injection results for the industrial program. The Δ-full variant shows very high fault coverage, with 3 SDCs in the case of transients and 0 in other cases. The Δ-stripped variant, however, results in a significant number of SDCs: the industrial program has a long and complex execution path such that errors do not propagate to the outputs. This is in contrast to HardCore where Δ-stripped had the same fault coverage as Δ-full. The reasons for SDCs are the same as for bubblesort and HardCore.

<table>
<thead>
<tr>
<th>DI</th>
<th>Δ-stripped</th>
<th>Δ-parallel</th>
<th>Δ-full</th>
<th>ANBD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.6</td>
<td>2.1</td>
<td>2.4</td>
<td>4.4</td>
<td>16.0</td>
</tr>
</tbody>
</table>

require decoding to the original values, their division and subsequent encoding (see Section IV-B).

Figs. 4–6 show the fault injection results for the industrial program. The Δ-full variant shows very high fault coverage, with 3 SDCs in the case of transients and 0 in other cases. The Δ-stripped variant, however, results in a significant number of SDCs: the industrial program has a long and complex execution path such that errors do not propagate to the outputs. This is in contrast to HardCore where Δ-stripped had the same fault coverage as Δ-full. The reasons for SDCs are the same as for bubblesort and HardCore.

E. Discussion

Δ-encoding was developed to provide a high level of fault tolerance. In this sense, we favor fault coverage over performance. Δ-encoded programs must be protected from all error types: transient, intermittent and permanent, single-bit and multiple-bit, single faults and multiple faults. Our experiments show that Δ-encoding (namely the Δ-full variant) achieves an average fault coverage of 99.997%.

We consider performance slowdowns of 3-4 times acceptable for our use cases. First of all, safety-critical computations are usually limited in size and not resource-demanding. Second, a software-only encoded processing approach is inherently slow, and a slowdown of several times is a significant improvement compared to the previous works on AN-encoding.
Unfortunately, we could not obtain the implementations of AN-encoding [11] or duplicated instructions [15]. However, we can perform an indirect comparison on the mutual quicksort benchmark to put δ-encoding into perspective (see Table II). The duplicated instructions approach (DI in the table) reveals a slowdown of 1.6x in the best case [15]; the ANBD-variant of AN-encoding has a slowdown of 16x [11]. δ-encoding shows performance numbers closer to duplicated instructions, with the slowdowns of 2–4x. This indicates that δ-encoding outperforms previous AN-encoding techniques, adding only a moderate overhead on top of duplicate execution.

For performance, our approach relies on deep instruction pipelining, out-of-order execution and sophisticated branch prediction in modern CPUs. All these techniques enable effective scheduling of instructions. Programs usually do not utilize instruction pipeline and branch prediction fully. δ-encoding takes advantage of an underutilized pipeline and branch predictor, such that the two copies of data can be processed in parallel. Table III shows that the number of instructions per cycle roughly doubles in δ-encoding programs, while the number of branch misses drops drastically. These numbers prove that δ-encoding benefits from heavily utilized pipeline and branch predictor.

VII. RELATED WORK

Local error detection research has a long history. It began in 1960s with pure hardware approaches used in highly available servers and space industry; starting from late 1990s, research focus shifted to software-only approaches, commonly known as software-implemented hardware fault tolerance (SIHFT).

A. Hardware-based approaches

Hardware-implemented error detection is exemplified by the evolution of two mainframe systems: IBM S/360 (now called IBM System z) and Tandem NonStop (now HP NonStop) [8]. These systems provide massive redundancy to achieve high availability: lockstepped proprietary CPUs, redundant CPU logic, ECC-protected memory and caches, and redundant hardware components and paths. The two systems guarantee very high fault coverage, but hardware implementation implies very high economic costs. δ-encoding can be seen as a much cheaper alternative to harden only a small subset of software stack run on commodity hardware.

A cost-effective hardware approach is to use simple checkers which observe activities of commodity hardware units and raise exceptions in case of errors. For example, the DIVA checker [27] commits CPU outputs only after it verified their correctness. Argus [28] implements four independent checkers to validate four CPU/memory tasks: control flow, data flow, computation, and memory accesses. Nostradamus [29] is yet another checker that compares an instruction’s expected impact on the CPU state to the actual impact on the state. Though the approaches incur low performance overhead (5-10%), they require significant changes in hardware, whereas δ-encoding is purely software-based and provides the same error detection guarantees.

Symptom-based detection (e.g., ReStore [30]) analyzes anomalous behavior of hardware such as memory access exceptions, mispredicted branches and cache misses. However, the approach cannot offer adequate fault coverage required in safety-critical systems, detecting only about a half of propagated faults.

B. Software-based approaches

Redundant Multithreading (RMT) [31] protects from transient faults by executing two copies of the program on two cores, periodically comparing their outputs. However, the technique assumes existence of a spare core, therefore typical embedded systems with single-core CPUs cannot benefit from RMT. In contrast, δ-encoding requires only one core for computations.

In duplicated instructions approach, program flow executes twice on the same core. The approach was first proposed in EDDI [15] and later refined in SWIFT [16]. Both solutions concentrate on transient errors and favor performance over fault coverage; moreover, SWIFT has an assumption of ECC-protected memory which does not hold for commodity and embedded hardware. Interestingly, EDDI’s offshoot called ED4I [32] is similar to δ-encoding: it combines data diversity and duplicated instructions, protecting from permanent faults. Unfortunately, ED4I was a theoretical attempt and was not even evaluated for performance, whereas δ-encoding is a complete and practical solution.

Encoded processing uses AN codes theory and was first used as a pure hardware approach; an example is a STAR computer designed for spacecrafts [33]. Forin [12] laid the foundations of software-implemented encoded processing, which was later extensively researched by Schiffel [11]. However, AN-encoding and variants thereof, which were used in these works, reveal imbalance in fault coverage versus performance: pure AN encoding has low fault coverage, ANB- and ANBD-variants have low performance. Our proposed δ-encoding provides balance between the two metrics.

VIII. CONCLUSION AND FUTURE WORK

We presented δ-encoding, a fault detection mechanism that covers not only commonly assumed Single Event Upsets, but also multiple-bit, intermittent and permanent faults. To achieve high fault coverage, δ-encoding combines two approaches: AN codes and duplicated instructions. As our evaluation shows, δ-encoding achieves fault coverage of 99.997% at the cost of an average slowdown of 2-4 times.

Our first prototype is a source-to-source transformer. In our future work, we would like to implement δ-encoding as a compiler plug-in. In this way, we will be able to perform
sophisticated data flow analysis to remove redundant accumulations and make the compiler $\Delta$-encoding-aware.

Another direction of future work is a software-hardware $\Delta$-encoding approach. Accumulations and checks can be moved out of the critical path and encapsulated in a separate hardware module. $\Delta$-encoding could also benefit from additional instructions in Instruction Set Architecture (ISA).

Another interesting implication of $\Delta$-encoding is the recovery ability. If a fault affected only one copy of data, it is detected via AN codes. The second copy of data can be used to recover the first copy, masking the fault, and the execution can continue. Such recovery schemes are in our future plans.

We envisage security-related applications of $\Delta$-encoding. Data diversity and the ability to use different pairs of $\Delta$s for different parts of a program could enable protection against malicious attacks. We will investigate a combined approach offering both fault tolerance and security in our future work.

ACKNOWLEDGMENT

The authors thank Oleksii Oleksenko for help with evaluation and Andreas Dixius for insightful suggestions. This work is partly supported by the German Research Foundation (DFG) within the Cluster of Excellence “Center for Advancing Electronics Dresden”.

REFERENCES


