Small Faults Grow Up - Verification of Error Masking Robustness in Arithmetically Encoded Programs

Anja F. Karl[1(✉)], Robert Schilling[1,2], Roderick Bloem[1], and Stefan Mangard[1]

[1] Graz University of Technology, Inffeldgasse 16A, 8010 Graz, Austria
{anja.karl,robert.schilling,roderick.bloem,stefan.mangard}@iaik.tugraz.at
[2] Know-Center GmbH, Inffeldgasse 13/6, 8010 Graz, Austria

Abstract. The increasing prevalence of soft errors and security concerns due to recent attacks like rowhammer have caused increased interest in the robustness of software against bit flips.

Arithmetic codes can be used as a protection mechanism to detect small errors injected in the program's data. However, the accumulation of propagated errors can increase the number of bits flips in a variable - possibly up to an undetectable level.

The effect of error masking can occur: An error weight exceeds the limitations of the code and a new, valid, but incorrect code word is formed. Masked errors are undetectable, and it is crucial to check variables for bit flips before error masking can occur.

In this paper, we develop a theory of provably robust arithmetic programs. We focus on the interaction of bit flips that can happen at different locations in the program and the propagation and possible masking of errors. We show how this interaction can be formally modeled and how off-the-shelf model checkers can be used to show correctness. We evaluate our approach based on prominent and security relevant algorithms and show that even multiple faults injected at any time into any variables can be handled by our method.

Keywords: Formal verification · Fault injection
Error detection codes · Arithmetic codes · Error masking

This project has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No. 681402), by the Austrian Science Fund (FWF) through the research network RiSE (S11406-N23), and by the Austrian Research Promotion Agency (FFG) via the competence center Know-Center, which is funded in the context of COMET Competence Centers for Excellent Technologies by BMVIT, BMWFW, and Styria. The authors would like to especially thank Karin Greiml and Bettina Könighofer for their support.

© Springer Nature Switzerland AG 2019
C. Enea and R. Piskac (Eds.): VMCAI 2019, LNCS 11388, pp. 183–204, 2019.
https://doi.org/10.1007/978-3-030-11245-5_9

1 Introduction

A typical assumption when writing software is that registers and memory content do not change unless the software performs a write operation on these locations. However, in practice, this assumption is challenged in several ways. On the one hand, the feature size of transistors in processors and memories keeps shrinking and shrinking, which allows natural phenomena like cosmic radiation to sporadically flip bits in memories and processors [4]. On the other hand, there exist attack techniques that aim at overcoming security mechanisms of systems by inducing targeted faults into a system. There is a wide range of publications on how to induce faults in systems using for example voltage glitches [3] or lasers [29]. The rowhammer effect [15] even allows attackers to cause bit flips remotely without any physical access to the target device.

Independent of whether a fault is caused by a natural phenomenon or an attacker, we refer to any change of a system state that is not caused by the software itself as a fault. Faults have huge implications on the security and safety of a system. Even a single bit flip, can lead to a critical system failure or reveal secret cryptographic keys (e.g. [1,7]). Consequently, appropriate mechanisms for detecting and handling faults are necessary.

The first error detection codes have been invented by Golay [13] and Hamming [14]. They proposed to add redundancy to every number, to increase the Hamming Distance [14] between encoded numbers. The higher the size of redundancy, the more bit flips can be detected. In the subsequent years, a special form of error detection codes have been discovered: Arithmetic codes do not only detect up to a fixed number of bit flips, the code words also remain valid over a certain set of arithmetic operations, e.g. $\texttt{encode}(a) +_{enc} \texttt{encode}(b) = \texttt{encode}(a + b)$. The number of detectable bit flips depends on the minimum arithmetic distance between valid code words [17], referred to as d_{min}. Examples for arithmetic error detection codes are AN, AN+B and residue codes [9,10,22].

1.1 Error Masking

In this work, we build up on the theory of arithmetic distance between arithmetic code words [17] and extend it to describe the propagation of errors and their arithmetic weights over an arithmetic program.

Listing 1.1. Copy of an invalid code word, resulting in two faulted variables `a` and `b`.

```
1 a := encode(0)
2 a := flip(a, 0th bit)
3 b := a
```

Every typical program contains data dependencies. If a value depends on a faulted one, it is influenced by that fault and is unlikely to be correct – the error propagated to the new variable. Listing 1.1 shows a simple example of an error propagating from one faulted variable to another one.

Listing 1.2. The sum of two invalid code words `a` and `b`, yields a faulted code word `c` containing two flipped bits.

```
1 a := encode(0)
2 a := flip(a, 0th bit)
3 b := a + a
4 c := a + b
```

As soon as an instruction has two faulted operands, the arithmetic weight of the errors can accumulate, and as a result the new error's weight can exceed the detection limit d_{min} of the code. In Listing 1.2, the flip of the 0th bit in `a` results in a flip of the 1st bit in `b`. Both errors accumulate to two bit flips in `c`.

Definition 1 (Error Masking). *Error masking is the effect of a new, valid, but incorrect code word emerging from an operation with two faulted operands.*

Listing 1.3. The injected fault is detected before errors can accumulate.

```
1 a := encode(0)
2 a := flip(a, 0th bit)
3 b := a + a
4 check(b)
5 c := a + b
```

A countermeasure for error masking is to check variables for errors at intermediate program locations, like in the example in Listing 1.3. However, it is non-trivial to determine where to place these checks: on the one hand, too many checks increase the run time of a program significantly, on the other hand, missing checks can lead to error masking.

1.2 Contribution

Within this work, we present a technique to prove that a program is robust against error masking. The following three points summarize our contribution:

1. We introduce the theory behind the effect of error masking based on the concept of error propagation over arithmetically encoded programs.
2. We use these insights to define the property of error masking robustness and present a novel technique to prove that the checks inside a program are sufficient to prevent error masking.
3. We demonstrate the capabilities of our approach based on real world programs. We were able to detect error masking vulnerabilities in cryptography algorithms and propose verifiable robust adaptions of these algorithms containing intermediate checks.

The core idea of our proposed method is the translation of an input program into a model of its worst-case error propagation, and to evaluate the model using an off-the-shelf model checker. With our method, we are not limited to detect robustness violations, but also receive indications of the problematic statements.

Furthermore, our approach is generic for all arithmetic encoding schemes, as long as there is a minimum arithmetic distance d_{min} between valid code words.

The flexibility of the technique allows us to use fault specifications of varying complexity. In contrast to other approaches, our method allows us to evaluate a program in the presence of *multiple faults* distributed over *all possible locations*!

1.3 Outline

The remainder of this paper is organized as follows: First, Sect. 2 describes the state of the art and related work. Next, Sect. 3 states the preliminaries and explains the concept of arithmetic codes and its most prominent examples. Our proposed approach to detect error masking is presented in Sects. 4 and 5: Sect. 4 describes the input language and the fault model, and Sect. 5 states the process to create a verifiable abstraction of the program under verification. Following, we prove the correctness of our approach in Sect. 6 and present our experimental results in Sect. 7. Finally, we conclude with a discussion of (dis-)advantages of our approach in Sect. 8 and a summary in Sect. 9.

2 Related Work

The first papers on arithmetic codes can be dated back to the 1950's and 1960's [9,10,17,22]. They describe a class of error detection codes that natively supports arithmetic operations without decoding the code word. While arithmetic codes have been developed to detect and correct bit flips during data transmission, they turned out to be also well suited as protection mechanism against a more recent concern: Using modern technology, adversaries are able to intentionally inject faults during program execution and thus reveal secret information [18].

In the recent years, researchers developed methods to automatically encode programs at compile time [11,25,26]. Although some of the required checks can be inserted automatically, they are insufficient for the prevention of error masking, and the user needs to specify further check locations himself. However, there is currently no exact theory to decide where necessary checks are required. This paper addresses this problem by introducing a method to automatically evaluate the placement of checks inside a program.

The idea of applying formal methods to verify the robustness of programs against faults is shared with multiple related papers: Pattabiraman et al. [21] and Larsson and Hähnle [16] both propose to use symbolic execution. The first of these two papers describes a method, where registers and memory locations are symbolically tagged with an *err* label, and error propagation is modelled through duplication of this label. The framework runs user defined error detectors to identify and report problems. However, the authors do not consider the exact number of bit flips on a variable, which prevents the tool from identifying error masking. The second publication focuses on the symbolic injection of multiple bit flips at fixed fault locations. In contrast to our work, it proposes

a method tailored to the principle of code duplication as countermeasure. This method compares the result of two versions of the same code, where one is based on faulted data. The effectiveness of code multiplication requires a strict independence of all redundant data paths. Walker et al. [31] introduce a method to identify such dependencies inside programs.

The idea of using LLVM bitcode transformations to add explicit fault injections to the source code is shared with the papers [30] and [12]. The idea of [30] is to execute two versions of a program - the original and a faulted version - and to evaluate user defined predicates. Every combination of the program counter and the state of these predicates form a node in a transition diagram. If an execution ever reaches a node unreachable in the fault-free transition diagram their tool reports an error. In the second paper, mutated binaries are model checked against a given specification. The results are then compared with the results of a fault-free verification run to identify differences. All those papers share similarities with our work, but they apply to different countermeasures and are not designed to detect error masking.

On the side of formal verification of programs using error detection codes, as to our knowledge, only few publications exist so far. Meola [20] formally proved the robustness of a small encoded program using Hoare Logic, and Schiffel [27] investigates the soundness and completeness of arithmetic codes using formal methods. Schiffel posits that the formal verification of AN-encoded programs using model checkers is impossible due to the exponential increase of verification time. We address this challenge by creating an abstraction of the program, only considering the error's weight instead of a variable's value.

3 Arithmetic Error Detecting Codes

Error detecting codes are a well-known way to detect errors during storage or computation. They can be divided into multiple sub-classes, among them the class of arithmetic error detection codes. These codes do not only guarantee a detection of all errors with an arithmetic weight smaller a constant d_{min}, they also remain valid over certain arithmetic operations, like additions.

3.1 Examples for Arithmetic Codes

One prominent example for an arithmetic code is the AN-code [9,10,26]. All valid AN code words are multiples of an user-defined constant A, with `encode`$(x) = x \cdot A$. To check a code word for validity, the remainder of the code word divided by A is calculated. For all valid code words, this remainder must be 0, otherwise the check detects an error and aborts execution. In the case of AN codes the check aborts, if a code word is not a multiple of A, $var_{enc} \bmod A \neq 0$.

A second class of arithmetic codes are residue codes [17]. A residue code word is defined by x concatenated with $x \bmod M$, given a constant modulus M, `encode`$(x) = (x \mid x \bmod M)$. This code separates the redundancy part from the functional value x, thus the name *separate code*. Although the robustness of the

code is defined by the modulus M, residue codes only guarantee detection of a single bit flip. To overcome this limitation, the redundancy part can be increased by using more than one residue [23,24], yielding a multi-residue code.

3.2 Arithmetic Weight and Distance

Both, AN-codes and (multi-) residue codes use the arithmetic weight and the arithmetic distance to quantify the robustness of the instantiated code. These properties are similar to the Hamming weight and Hamming distance [14] used for binary linear codes. The arithmetic weight $W(|x|)$ of the integer value x is defined as the minimum number of non-zero coefficients in the signed digit representation of x.

$$W(|x|) = \min \left\{ \sum_{i=0}^{\infty} |b_i| \;\middle|\; b_i \in \{-1, 0, 1\},\, x = \sum_{i=0}^{\infty} b_i 2^i \right\}$$

The arithmetic distance $d(x_1, x_2)$ between the two integers x_1 and x_2 is equal to the arithmetic weight of the absolute difference between x_1 and x_2.

$$d(x_1, x_2) = W(|x_1 - x_2|)$$

The constant d_{min} is the only information about the encoding our method requires. It is defined as the minimum arithmetic distance between any two valid code words x_{enc1} and x_{enc2}. All errors with a weight up to d_{min} are guaranteed to be detected by a properly implemented check. This property is essential to verify the error masking robustness, as described in the subsequent sections.

$$d_{min} = \min_{x_{enc1} \neq x_{enc2}} d(x_{enc1}, x_{enc2})$$

4 Error Masking Robust Programs

In this section, we first describe the input program's language and define the fault model considered in our approach. Next, we explain, how to derive a program P_f containing explicit fault injections. Finally, we present a formal definition of robustness against error masking based on an explicitly faulted program P_f.

4.1 Programs

Our robustness verification method is applicable for arithmetic programs of the following form.

Definition 2 (Input Programs). *An input program P is a directed graph $P = (V, E, \lambda, v_0, Var)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, $\lambda : V \rightarrow S$ is a mapping of vertices to statements, $v_0 \in V$ is a start vertex, and $Var = Var^{loc} \cup Var^{arg}$ is a set of local variables and program arguments.*

All variables $var \in Var$ and constants $const_{enc} \in Const_{enc} = \{\texttt{encode}(n) \mid n \in \mathbb{N}\}$ are arithmetically encoded natural numbers. All statements $s \in S$ are either arithmetic instructions $s \in S_{arith}$ or control-flow directives $s \in S_{cf}$; i.e., $S = S_{arith} \cup S_{cf}$. Arithmetic instructions $s \in S_{arith}$ can either be assignments of constants $s \in S_{assign}$, additions $s \in S_{add1} \cup S_{add2}$, or subtractions $s \in S_{sub}$. We distinguish between additions with the same variable for both operands, $s \in S_{add1}$, and additions with different variables, $s \in S_{add2}$. Formally, we have $S_{arith} = S_{assign} \cup S_{add1} \cup S_{add2} \cup S_{sub}$, with

$$S_{assign} = \{var := const_{enc} \mid var \in Var, const_{enc} \in Const_{enc}\},$$
$$S_{add1} = \{var := var_1 + var_1 \mid var, var_1 \in Var\},$$
$$S_{add2} = \{var := var_1 + var_2 \mid var, var_1, var_2 \in Var\}, \text{ and}$$
$$S_{sub} = \{var := var_1 - var_2 \mid var, var_1, var_2 \in Var\}.$$

Control-flow directives $s \in S_{cf}$ include direct jumps $s \in S_{jump}$, conditional branches $s \in S_{cbranch}$, checks $s \in S_{check}$ and terminators $s \in S_{ret}$. We have $S_{cf} = S_{jump} \cup S_{cbranch} \cup S_{check} \cup S_{ret}$, with

$$S_{jump} = \{\texttt{goto}\ v \mid v \in V\},$$
$$S_{cbranch} = \{\texttt{if}\ (c)\ \texttt{goto}\ v_1\ \texttt{else goto}\ v_2 \mid v_1, v_2 \in V, c \in C\},$$
$$S_{check} = \{\texttt{check}\ (var) \mid var \in Var\}, \text{ and}$$
$$S_{ret} = \{\texttt{return}\ var \mid var \in Var\}.$$

Boolean conditions $c \in C$ are either comparisons val_1 `op` val_2, with $val_1, val_2 \in Const_{enc} \cup Var$ and `op` $\in \{<, \leq, =, \neq, \geq, >\}$, or boolean combinations of comparisons. In the fault-free case, conditional branches continue with the first target vertex, if the condition c evaluates to true, and with the second vertex otherwise. Every conditional branch performs an implicit check on all operands in c. To avoid flipping the boolean value of c itself, we propose to use branch protection algorithms like [28]. The execution of a conditional branch can fall into one of three cases: (1) Every operand is correct and the execution jumps to the correct vertex. (2) Any operand in the condition is faulted, but contains a detectable fault. In this case, the conditional branch statement aborts execution and enters a safe state. (3) The error weight on the compared operands exceeds $d_{min} - 1$, and the branch protection mechanism can miss the fault. The statement continues with either of both `goto` statements and executes a possibly invalid path. This behavior is a consequence of error masking and will be detected by our method.

A runtime assertion `check` (var) checks a code word var for validity, aborts execution and enters a safe state if it detects a fault on this variable. However, checks are not able to detect masked errors and only guarantee to disclose errors with a maximum arithmetic weight of $d_{min} - 1$. The actual implementation of a check depends on the encoding scheme of the program and is both possible in hardware and in software.

Every vertex v_i with a statement $\lambda(v_i) \in S_{arith} \cup S_{check} \cup S_{jump}$ has exactly one successor v_{i+1}. If $\lambda(v_i) =$ `goto` v_j, the destination vertex v_j must

be the single successor of v_i. All vertices v_i with conditional branch statements $\lambda(v_i) = \texttt{if}\ (c)\ \texttt{goto}\ v_j\ \texttt{else goto}\ v'_j$ have exactly two outgoing edges to v_j and v'_j, and all vertices v_i with return statements $\lambda(v_i) \in S_{ret}$ have zero successors.

Our method requires the whole program to be encoded using the same encoding scheme and the same encoding constants. As a consequence, there is a value $d_{min} > 1$, which is smaller or equal to the arithmetic distance of any two valid code words. The constant $d_{min} - 1$ forms the upper limit for the number of guaranteed detectable bit flips and needs to be known in order to evaluate a program using our method. The programmer is responsible for choosing an appropriate encoding scheme, such that all operations in the program are possible in the encoded domain and no overflows can occur.

Listing 1.4. Running example.

```
1 toy():
2   a := encode(0)
3   b := a + a
4   check(b)
5   c := a + b
6   return c
```

As running example we use our small toy program from Listings 1.2 and 1.3. The `flip` in both programs was not intended and occurred due to either an attacker or environmental influences during execution. Listing 1.4 shows the original program, as it was written by the programmer.

4.2 Fault Model

This work focuses on faults in memory, where bits of variable values are flipped. Every fault consists of a (possibly negative) error Err of an arithmetic weight $W(|Err|) < d_{min}$ added to a variable var at any point in time during program execution. A special case of faults are bit flips. A single bit flip in the ith bit corresponds to an error $Err = b_i 2^i$, with $b_i = 1$ if the flip sets the bit, and $b_i = -1$ otherwise. Therefore, the arithmetic weight of a single bit flip is $W(|Err|) = 1$. All faults injected into a variable var remain present until a new value is assigned to var and overwrites the fault. In this work, we do not consider control-flow attacks as there are already promising countermeasures [28,32] to protect this attack vector. We assume that an integrity mechanism is present such that all instructions as well as the control-flow of the program are protected.

4.3 Explicitly Faulted Programs

In order to verify the robustness of a program, we need to make faults in the input program visible to the model checker. Therefore, we define a derived program with explicit fault injections. The derived program contains a copy of every vertex $v \in V$ called v'_f with the same statement; i.e., $\lambda_f(v'_f) = \lambda(v)$. Additionally, we add a vertex v''_f before every v'_f. The statement of v''_f injects faults explicitly into the operands of the statement $\lambda_f(v'_f)$. Formally, we define P_f as:

Definition 3 (Explicitly Faulted Program P_f). *Let $P = (V, E, \lambda, v_0, Var)$ be a program, let $V'_f = V \times \{1\}$ and $V''_f = V \times \{2\}$ be two copies of V, and let $V_f = V'_f \cup V''_f$. The explicitly faulted program $P_f = (V_f, E_f, \lambda_f, v_{0f}, Var_f)$ is a graph, where $E_f = E_{f_1} \cup E_{f_2}$ is the set of edges with $E_{f_1} = \{(v''_f, v'_f) \mid v''_f = (v, 2), v'_f = (v, 1), v \in V\}$ and $E_{f_2} = \{(v'_{1_f}, v''_{2_f}) \mid v'_{1_f} = (v_1, 1), v''_{2_f} = (v_2, 2), (v_1, v_2) \in E\}$, and $Var_f = Var$ is the set of variables. The start vertex v_{0f} is defined by $v_{0f} = (v_0, 2)$ and the statement function λ_f as*

$$\lambda_f((v,i)) = \begin{cases} \lambda(v) & \text{if} \quad i = 1 \\ var := var + Err_v & \text{if} \quad i = 2 \text{ and } \lambda(v) = \texttt{return}\ var \\ \begin{array}{l} var_1 := var_1 + Err1_v \\ var_2 := var_2 + Err2_v \end{array} & \text{if} \quad i = 2 \text{ and } \lambda(v) = var := var_1 \pm var_2 \\ \epsilon & \text{else.} \end{cases}$$

In this formula, Err_v denotes the error injected before execution of the statement of v into its operand. In the case of two operands, the $Err1_v$ is the error injected into the first operand and $Err2_v$ is the error injected into the second operand. If $\lambda(v)$ has no operands, the statement $\lambda_f((v, 2))$ is empty. The explicitly faulted version of our toy example is depicted in Listing 1.5.

Listing 1.5. P_f of the running example in Listing 1.4.

```
1   toy():
2       a := encode(0)
3
4       a := a + Err1_v1
5       a := a + Err2_v1
6       b := a + a
7
8       check(b)
9
10      a := a + Err1_v3
11      b := b + Err2_v3
12      c := a + b
13
14      c := c + Err_v4
15      return c
```

4.4 Robustness Condition

The explicit faults in P_f allow us to name the errors on every variable during execution. Therefore, we can introduce the following terms and define the condition for robustness of a program against error masking.

Definition 4 (Execution Path). *A path $\pi = \pi[0], \ldots, \pi[n]$ is a sequence of $n + 1$ vertices with $\pi[i] \in V$, where the program graph P has a directed edge between any two subsequent elements $(\pi[i], \pi[i + 1]) \in E$.*

Definition 5 (Execution Trace). *An execution trace* $\pi^{exec} = \pi[0], \ldots, \pi[n]$ *of a program* P *is an execution path through the program starting at* $\pi[0] = v_0$ *and ending with a vertex* $\pi[n]$*, with* $\lambda(\pi[n]) \in S_{ret}$*.*

Definition 6 (Feasible Execution Trace). *An execution path* π *is contained in an execution trace* π^{exec}*, if all elements of* π *are also included in* π^{exec} *and their order is preserved. An execution trace* π^{exec} *of a program* P *is feasible in an explicitly faulted program* P_f*, iff there is an execution trace* π_f^{exec}*, such that* π^{exec} *is contained in* π_f^{exec}*.*

Definition 7 (Fault-Free Program). *Given a program* P_f*, the fault-free program* P_f^0 *is defined as* P_f *with no errors injected at any vertex, i.e. for all* $v \in V$ *it holds that* $Err_v = 0$*,* $Err1_v = 0$*, and* $Err2_v = 0$*.*

Definition 8 (Program State). *Given a deterministic, explicitly faulted program* P_f *and fixed values for every program argument and injected errors, there is only one feasible execution trace* π*. We define the program state* $\Pi[t]$ *of* π *as the mapping from all variables to their value at execution step* t*. The function* $[\![\Pi[t] \mid var]\!]$ *returns the value of the variable var in this execution state, and* $[\![\Pi[t]]\!]_\pi$ *returns the execution path* $\pi[0], \ldots, \pi[t]$ *up to* $\pi[t]$*.*

Definition 9 (Error on a variable). *Given an execution state* Π_f *of* P_f *and the corresponding execution state* Π_f^0 *of* P_f^0*, the error* $[\![\Pi_f[t] \mid Err(var)]\!]$ *on a variable var is the difference between* $[\![\Pi_f[t] \mid var]\!]$ *and* $[\![\Pi_f^0[t] \mid var]\!]$*.*

Definition 10 (Robustness of an explicitly faulted program). *A faulted program* P_f *is error masking robust if every feasible execution trace is also feasible in the fault-free program* P_f^0 *and all its executions return either a fault-free value* $[\![\Pi_f[k] \mid Err(var)]\!] = 0$ *or any fault on the returned value* $[\![\Pi_f[k] \mid var]\!]$ *is smaller than* d_{min} *and therefore guaranteed detectable.*

Definition 11 (Robustness of an program). *A program* P *is robust against error masking iff the explicitly faulted program* P_f *is robust against error masking.*

To guarantee the robustness against error masking, the properties stated in Definition 10 are required to hold on the explicitly faulted program. The first condition can be ensured by preventing error masking on any variables compared in a branch condition, while the latter requires the absence of error masking on the return value. Both problems are detected by the method described in the next section.

5 Proving a Program Robust Against Error Masking

This section describes the verification of the error masking robustness of a program, as defined in Sect. 4. Figure 1 depicts the verification process: starting from an input program P, we create the explicitly faulted program P_f and derive an abstract model of the worst case error weight propagation P_w. This model is

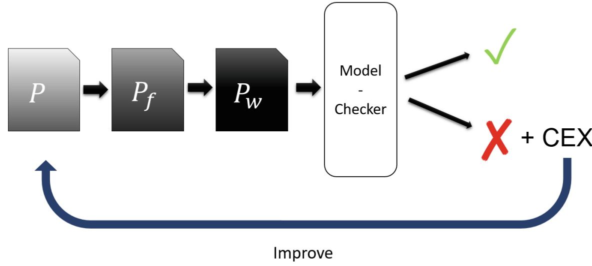

Fig. 1. The work flow of the verification process.

then model checked for error masking robustness. In the case of error masking possibilities, the model checker generates a counterexample, which can be used to improve P by inserting additional checks. If the model checker reports no errors, the program is guaranteed to be error masking robust.

The main idea behind our method is to track the maximum error weight on each variable and to ensure this error weight never exceeds $d_{min} - 1$. In this case, errors can never mask each other and are always detectable. Our technique to prove error masking robustness involves three main steps: (1) We derive the explicitly faulted program $P \rightsquigarrow P_f$ from the input program P, as described in Sect. 4. (2) We transforms the faulted program P_f into an error weight counting program $P_f \rightsquigarrow P_w$. The program P_w is a model of the worst case error weight propagation and contains assertions for ensuring P to be robust. (3) We apply an off-the-shelf model checker to evaluate the new program P_w. The model checker proves the absence of error masking or provides a counterexample in case of any violations of the robustness assertions.

In order to define the error weight counting program P_w, we first introduce the concept of fault specifications and afterwards explain the language of P_w and its construction.

5.1 Fault Specification

The fault specification FS constraints the maximum arithmetic weight of any injected error and is provided by the user.

Definition 12 (Maximum Injected Error Weight). *The maximum injected error weight W_v denotes the maximum weight of errors injected over all visits to a vertex v into the operand of $\lambda(v)$. In the case of two operands, $W1_v$ and $W2_v$ are the maximum injected error weights of the first and the second operand.*

Definition 13 (Fault Specification). *A fault specification FS is a Boolean expression over predicates $\sum(W_v)$ $\texttt{op}$ n, with $\texttt{op} \in \{<, \leq, =, \geq, >, \neq\}$ and a constant $n \leq d_{min}$, such that FS restricts every injected error weight to an upper limit of $d_{min} - 1$.*

A simple example for a fault specification is to limit the sum of all maximum injected error weights to a constant $n < d_{min}$; i.e.: $\sum W_v + \sum W1_v + \sum W2_v \leq n$.

5.2 Adaption of the Input Language

The language of P_w is defined as follows. Let Var_w be a copy of all variables of Var. For every node $v \in V$ we have an error weight injection variable W_v for each operand of $\lambda(v)$. Similar to the statements S of P, we define the statements S_w of P_w as combination of arithmetic instructions S_{arith_w} and control-flow directives S_{cf_w}. In the case of P_w, the arithmetic statements include the initialization of an error weight inject $S_{init_inj_w}$, the deletion of an error weight S_{zero_w}, the duplication of an error weight S_{dupl_w} and the addition of two error weights S_{add_w}. Formally, these statements are defined as:

$$
\begin{aligned}
S_{init_inj_w} &= \{W_v := * \mid v \in V\}, \\
S_{zero_w} &= \{var_w := 0 \mid var_w \in Var_w\}, \\
S_{dupl_w} &= \{var_w := var_{1w} \mid var_w, var_{1w} \in Var_w\}, \text{ and} \\
S_{add_w} &= \{var_w := var_{1w} + var_{2w} \mid var_w, var_{1w}, var_{2w} \in Var_w\}.
\end{aligned}
$$

Control-flow directives S_{cf_w} include jumps S_{jump_w}, conditional branches $S_{cbranch_w}$, terminators S_{ret_w}, assertions S_{assert_w} and assumptions S_{assume_w}, i.e. $S_{cf_w} = S_{jump_w} \cup S_{cbranch_w} \cup S_{ret_w} \cup S_{assert_w} \cup S_{assume_w}$. Let V_w be the set of vertices in P_w, Var_w a set of variables, and fs a fault specification. We can define the different kinds of control-flow directives of P_w as:

$$
\begin{aligned}
S_{jump_w} &= \{\texttt{goto}\ v_w \mid v_w \in V_w\}, \\
S_{cbranch_w} &= \{\texttt{if}\ (*)\ \texttt{goto}\ v_{1w}\ \texttt{else goto}\ v_{2w} \mid v_{1w}, v_{2w} \in V_w\}, \\
S_{ret_w} &= \{\texttt{return}\}, \\
S_{assert_w} &= \{\texttt{assert}\,(var_w < d_{min}) \mid var_w \in Var_w\}, \text{ and} \\
S_{assume_w} &= \{\texttt{assume}\,(var_w == 0) \mid var_w \in Var_w\} \cup \{\texttt{assume}\,(FS)\}
\end{aligned}
$$

In this syntax, the $*$ symbol denotes non-deterministic value. The task of the model checker is to prove that for any value as $*$ the assertions inside P_w are never violated, given that all assumptions are fulfilled.

5.3 Translation of the Explicitly Faulted Program into a Weight Counting Program

The error weight counting program P_w can be derived from an explicitly faulted program P_f, via the transformation $P_f \rightsquigarrow P_w$. P_w is an abstraction of the program P_f, which stores only the upper bound of the error weight on the corresponding variables' value. Therefore, P_w contains one error weight counter $var_w \in Var_w$ for every variable of Var_f. All error weight counters in $Var_w = Var_f$ are unsigned variables, which are initialized to zero. In addition to the two copies of V in P_f, P_w contains a third copy $V_w''' = V \times \{3\}$, where assertions are added. Furthermore, P_w starts with multiple initialization vertices, namely $v_w^{ew_init}$, $v_w^{W_init}$, and v_w^{fs}. The vertex $v_w^{ew_init}$ is the first vertex of the program with the following statements:

$$\lambda_w(v_w^{ew_init}) = \{var_w := 0 \mid var_w \in Var_w\}.$$

Next, within the node $v_w^{W_init}$, every maximum injected error weight W_v is set to a non-deterministic, positive integer:

$$\lambda_w(v_w^{W_init}) = \{W_v := * \mid v \in V\} \cup \{W1_v := * \mid v \in V\} \cup \{W2_v := * \mid v \in V\}.$$

As final initialization step, the node v^{fs} limits the maximum injected error weights according to the fault specification:

$$\lambda_w(v_w^{fs}) = \texttt{assume}(fs).$$

Let v'_f be a vertex in V'_f, and v'_w be the corresponding copy in V'_w. Furthermore, let each var_w be the error weight counter for the variable var_f. Every arithmetic statement $\lambda_f(v'_f) \in S_{arith}$ is transformed into a new statement $\lambda_w(v'_w)$ by the following rules:

$$\lambda_w(v'_w) = \begin{cases} var_w := 0 & \text{if } \lambda_f(v'_f) = var := const_{enc} \\ var_w := var_{1w} & \text{if } \lambda_f(v'_f) = var := var_1 + var_1 \\ var_w := var_{1w} + var_{2w} & \text{if } \lambda_f(v'_f) = var := var_1 + var_2 \\ var_w := var_{1w} + var_{2w} & \text{if } \lambda_f(v'_f) = var := var_1 - var_2 \end{cases}.$$

Assigning a constant to a variable var_f is equivalent to erasing the error that was stored in var_f before execution of the assignment. Therefore, the error weigh counter is erased. When the same variable is added to itself, the error itself is multiplied by two, but its weight remains the same. Therefore, the addition of the same variables $var := var_1 + var_1$ is the same as copying the error weight counter var_{1w} to var_w. Finally, every addition and subtraction has the worst case error propagation $var_w := var_{1w} + var_{2w}$, as modelled by the last two cases.

Let c_{iw} be all operands of a condition c. Similarly to $\lambda_f(v'_f) \in S_{arith}$, every control-flow directive $\lambda_f(v'_f) \in S_{cf}$ is translated according to

$$\lambda_w(v'_w) = \begin{cases} \texttt{goto } v_{1w} & \text{if } \lambda_f(v'_f) = \texttt{goto } v_{1f} \\ \begin{array}{l}\texttt{assume}\,(c_{iw} = 0) \\ \texttt{if}\,(*)\ \ \texttt{goto } v'_{1w} \\ \texttt{else}\ \ \texttt{goto } v'_{2w}\end{array} & \text{if } \lambda_f(v'_f) = \begin{array}{l}\texttt{if}\,(c)\ \texttt{goto } v'_{1f} \\ \texttt{else goto } v'_{2f}\end{array} \\ \texttt{assume}\,(var_w = 0) & \text{if } \lambda_f(v'_f) = \texttt{check}\,(var) \\ \begin{array}{l}\texttt{assert}\,(var_w < d_{min}) \\ \texttt{return}\end{array} & \text{if } \lambda_f(v'_f) = \texttt{return } var_f \end{cases}.$$

Every unconditional jump in P_f corresponds to the same jump in P_w. However, every conditional branch is transformed into a non-deterministic branch, regardless of the previous branch condition. This transformation guarantees independence of actual variable values and brings along both advantages and restrictions. These matters are further discussed in Sect. 8. As all variables accessed by c are

implicitly checked by a branch protection algorithm as described in Subsect. 4.1, the new statement begins with the assumptions that all c_{iw} are fault-free. When a `check`(var) statement of P_f is executed, exactly one of the following cases must apply:

1. $0 < var < d_{min}$: In this case, an error is detected for sure and the execution is aborted. There cannot be any further error masking and therefore this case can be neglected.
2. $var \geq d_{min}$: In this case the program could either be terminated or continued. This case violates the robustness property and is reported by the assertion `assert`($var_w < d_{min}$).
3. $var = 0$: The only remaining case is the error free case, which can be assumed, once the robustness assertion has been passed.

Eventually, a return statement quits execution of a program and no further error masking can occur. Every return in P_f corresponds to a return in P_w.

Like in P_f, all fault injections are explicit. A fault injection in P_w is represented by an increment of the error weight counter by the maximum injectable error weight. After the error has been injected, there are no bit flips left for this location and the remaining error weight is set to 0.

$$\lambda_w(v''_w) = \begin{array}{l} var_w := var_w + W_v \\ W_v := 0 \end{array} \qquad \text{if } \lambda_f(v''_f) = var_f := var_f + Err_v$$

Finally, a model checker requires a definition of the correctness for a program. As defined in Definition 10, the correctness of the program can be guaranteed if all variables' error weights remain below d_{min}. If there is any chance this property is ever violated, the model checker should prompt a warning and give a violating counterexample. Within the program P_w, the correctness is assured by calls to the `assert` function. Let $v'''_w \in V'''_w$ be a node of the third vertex copy of V, and var_w be the error weight counter modified by $\lambda_w(v''_w)$. Then $\lambda_w(v'''_w)$ is given as

$$\lambda_w(v'''_w) = \texttt{assert}\,(var_w < d_{min} \mid var_w \in Var_w).$$

Given the previously defined construction, we can define P_w as follows.

Definition 14 (Error Weight Counting Program P_w). *Let* $V_w = \{v_w^{ew_init}, v_w^{W_init}, v_w^{fs}\} \cup V'_w \cup V''_w \cup V'''_w$ *be a set of vertices and* $E_w = \{(v_w^{ew_init}, v_w^{W_init}),\ (v_w^{W_init}, v_w^{fs})\} \cup \{(v''_w, v'_w) \mid v'_f \in V_f\} \cup \{(v'_w, v'''_w) \mid v_f \in V_f\} \cup \{(v'''_{1w}, v''_{2w}) \mid v'''_{1w} = (v_1, 3), v''_{2w} = (v_2, 2), (v_1, v_2) \in E_f\}$ *a set of edge between the nodes. Then* P_w *is defined as* $P_w = (V_w, E_w, \lambda_w, v_{0w}, Var_w)$*, with* $v_{0w} = v_w^{ew_init}$ *and* $Var_w = Var$.

After performing the steps described above, the transformation is complete. The resulting program P_w models the worst case error propagation and any potential error masking in P is present as an assertion violation in P_w.

Listing 1.6. P_w of the toy example.

```
1  toy():
2      a, b, c := 0
3      W1_v1, W2_v1, W1_v3, W2_v2, W^v4 := *
4      assume(W1_v1 + W2_v1 + W1_v3 + W2_v2 + W^v4 ≤ 2)

6      a := 0
7      assert(a < d_min)

9      a := a + W1_v1
10     a := a + W2_v1
11     b := a
12     assert(b < d_min)

14     assume(b = 0)

16     a := a + W1_v3
17     b := b + W2_v3
18     c := a + b
19     assert(c < d_min)

21     c := c + W_v4
22     assert(c < d_min)
23     return
```

The weight counting program of our toy example can be seen in Listing 1.6. Within the first line, it sets every error weight counter (`a, b,` and `c`) to zero. The next line initializes all error weight injections to arbitrary values before they are restricted according to the fault specification, in this case to at most two bit flips in total. The next lines (lines 6–22) consist of each the injection of the error weight into the operands, followed by the error propagation and the robustness assertions. The check on `b` in the middle of the program has been transformed to an `assume` and finally P_w ends with the transformed `return` statement.

5.4 Applying a Model Checker to Prove Correctness

As third step, we use a model checker to verify the resulting program P_w. For our running example, we are able to verify its error masking robustness, giving the fault specification $\sum(W_v) \leq 2$ with $d_{min} = 3$. However, without the line `check(b)`, the model checker successfully reports a vulnerability within the instruction `c := a + b`, if `a` contains an error of weight 2. This result corresponds to the expected outcome as illustrated in Sect. 1.

The next section will give a proof of correctness of our method, followed by an evaluation of the method using real world examples.

6 Proof of Correctness

We can show that for every potential error masking in P, P_w contains an assertion violation. For this, we use the following definitions.

Definition 15 (Mapping of a Program State). *Given a program state $\mathit{\Pi}_f[t]$ of the explicitly faulted program P_f, we define $\mathit{\Pi}_w(\mathit{\Pi}_f[t])$ as the corresponding program state of P_w, where for all Err_v it holds that $W(|[\![\mathit{\Pi}_f[0] \mid Err_v]\!]|) = [\![\mathit{\Pi}_w(\mathit{\Pi}_f[0]) \mid W_v]\!]$ and $[\![\mathit{\Pi}_w(\mathit{\Pi}_f[t])]\!]_\pi$ is the smallest execution trace containing $[\![\mathit{\Pi}_f[t]]\!]_\pi$.*

Theorem 1. *Let $\mathit{\Pi}_f[t]$ be a program state, where every variable is smaller or equal to its corresponding error weight counter in $\mathit{\Pi}_w(\mathit{\Pi}_f[t])$. After any statement $\lambda_f(v_f) \in S_{arith}$, the error of the variable var_f modified by $\lambda_f(v_f)$ is smaller or equal to the error weight counter var_w belonging to this variable.*

Proof. All arithmetic statements fall into one of the following cases: (1) In the case of $\lambda_f(v_f) = var_f := \texttt{encode}(c)$, a variable is set to a encoded constant, which originally contains no fault. $W(|Err(\texttt{encode}(c))|) = 0 \rightarrow var_w = 0 \geq W(|Err(var_f)|)$. (2) In the case of addition of the same variable with itself, $\lambda_f(v_f) = var_f := var_{f_1} + var_{f_1}$, we get $D(var_{f_1} + var_{f_1}, var^0_{f_1} + var^0_{f_1}) = W(|2Err(var_{f_1})|) = W(|Err(var_{f_1})|)$, such that $var_w = W(|Err(var_{f_1})|) = W(|Err(var_f)|)$. (3) If two different variables are added or subtracted, $\lambda_f(v_f) = var_f := var_{f_1} \pm var_{f_2}$, the new error weight fulfills the following inequality: $D(var_{f_1} + var_{f_2}, var^0_{f_1} + var^0_{f_2}) = W(|Err(var_{f_1}) - Err(var_{f_2})|) \leq W(|Err(var_{f_1})|) + W(|Err(var_{f_2})|)$. Therefore it holds that $var_w = W(|Err(var_{f_1})|) + W(|Err(var_{f_2})|) \geq W(|Err(var_f)|)$.

Theorem 2. *In any program state $\mathit{\Pi}_f[t]$ of P_f with $\mathit{\Pi}_w(\mathit{\Pi}_f[t])$ fulfilling all assumed conditions, the error of a variable $[\![\mathit{\Pi}_f[t] \mid Err(var_f)]\!]$ has at most the arithmetic weight stored in the corresponding error weight variable, i.e., var_w, $[\![\mathit{\Pi}_f[t] \mid Err(var_f)]\!] \leq [\![\mathit{\Pi}_w(\mathit{\Pi}_f[t]) \mid var_w]\!]$.*

Proof. Every execution trace π_f starts with the same vertex $\pi_f[0] = v_{0_f}$, where no errors could have been injected yet. Therefore, it is correct to assume that all variable's error weight are 0. Suppose all error weights in every program state $\mathit{\Pi}_w(\mathit{\Pi}_f[i])$ with $i < t$ are correct. $\forall i < t. \forall var_f [\![\mathit{\Pi}_f[i] \mid Err(var_f)]\!] \leq [\![\mathit{\Pi}_w(\mathit{\Pi}_f[i]) \mid var_w]\!]$. We can show that after any further step with $\pi_f[t+1] = v_f$, the variable modified by $\lambda_f(v_f)$ has an error weight $[\![\mathit{\Pi}_f[t+1] \mid Err(var_f)]\!] \leq [\![\mathit{\Pi}_w(\mathit{\Pi}_f[t+1]) \mid var_w]\!]$: The statement $\lambda_f(v_f)$ can be either an arithmetic statement, an control-flow directive or an error injection. Theorem 1 proves that this property is fulfilled for every statement $\lambda_f(v_f) \in S_{arith}$. In contrast to that, control-flow directives do not modify the error weights directly. As long as the execution follows the same path through the program $\forall t \mathit{\Pi}_f[t] = w(\mathit{\Pi}_f[t])$, the control-flow directives will not influence any error weights. Finally, given Definition 15 defines that all for all E_v it holds that $W(|[\![\mathit{\Pi}_f[0] \mid E_v]\!]|) = [\![\mathit{\Pi}_w(\mathit{\Pi}_f[0]) \mid W_v]\!]$. This guarantees that $[\![\mathit{\Pi}_f[t+1] \mid Err(var_f)]\!] \leq [\![\mathit{\Pi}_w(\mathit{\Pi}_f[t+1]) \mid var_w]\!]$.

This shows, that the weight of the error on all variables remains smaller or equal the value of the corresponding weight variables.

Theorem 3 (Transformation of Checks). *Every passed $\texttt{check}(var_f)$ either implies a violation of the assertion $\texttt{assert}(var_w < d_{min})$ or that $Err(var_f) = 0$.*

Proof. There are three cases for the execution of every check:

1. $0 < W(|Err(var_f)|) < d_{min}$: In this case, the check is not passed and the execution is aborted. No further error masking can occur.
2. $W(|Err(var_f)|) \geq d_{min}$: If the error weight exceeds the minimum arithmetic distance, Theorem 2 proves that $var_w \geq W(|Err(var_f)|)$, and the assertion `assert`$(var_w < d_{min})$ is violated.
3. $W(|Err(var_f)|) = 0$: The only remaining case is the error free case, which can be assumed, once the robustness assertion has been passed.

Theorem 4. *Given a program P_w containing loops, where all error weights are injected in the first iteration, and a program P'_w abstracting the same program P, with all error weight injections distributed over all infinite loop iterations, it is always true that if P_w is correct, then P'_w also is correct.*

Proof. The value of an error weight counter in a program state $\Pi_w[t]$ can be represented as the sum of multiple error weight injections. $[\![\Pi_w[t] \mid var_w]\!] = \sum_{j=0}^{\infty} k_v[j]W_v[j]$, where the factor k_v indicates the number of times the injected error weight has accumulated in an error weight counter, and $W_v[j]$ is the error weight injected in loop iteration j. In the case of P_w, $W_v[0] = W_v$ and $\forall j > 0 : W_v[j] = 0$, while all $W_v[j]$ of P'_w are smaller or equal those of P_w. Furthermore, $\forall j > 0 : k_v[0] = 0 \vee k_v[0] > k_v[j]$, therefore, the only way that $[\![\Pi_w[t] \mid var_w]\!] < [\![\Pi'_w[t] \mid var_w]\!]$ can be achieved is, if var_w is overwritten after injecting $W_v[0]$ ($k_v[0] = 0$), and j is the current loop iteration. However, in the next loop iteration, this error weight will be overwritten again ($k_v[j] = 0$). The maximum value during the first loop iteration will never be exceeded.

Theorem 5 (Correctness of P_w). *If P_w is correct, P_f is correct and P is robust against error masking.*

Proof. Assume P_f is incorrect. Let $\Pi_f[k]$ be the last execution state of a program run violating the correctness of P_f, and var_{ret} be the returned value. A program run Π_f can violate the correctness condition in two ways: (1) The return value is a faulted code word $[\![\Pi_f[k] \mid var_{ret}]\!] \neq [\![\Pi_f^0[k] \mid var_{ret}]\!]$, with its error weight undetectable $[\![\Pi_f[k] \mid W(|Err(var_{ret})|)]\!] \geq d_{min}$, or (2), an invalid path through the program is taken. In case (1), Theorem 2 provides a proof, that $[\![\Pi_f[k], Err(var_{ret})]\!] > d_{min} \rightarrow [\![\Pi_w(\Pi_f[k]) \mid var_{wret}]\!] > d_{min}$. Therefore, at least the last assertion in P_w is violated and P_w is incorrect. Case (2) can only be caused, if the execution of a statement of the form `if` $(cond)$ `goto` v_{1_f} `else goto` v_{2_f} continues with the wrong branch. An appropriate branch protection mechanism will abort execution as long as it detects any fault in either the compared operands or in the comparison result. This leaves the remaining situations where (2) is possible, as those, where a fault on the comparison operands contains a masked error. However, Theorem 2 proves that the assertions in P_w detect this case as well, and therefore P_w is incorrect in this case too. This shows, that any violation of P_f will always result in a violation of P_w, and if P_w is correct, that implies that P_f is robust.

Theorem 6 (Decidability). *The correctness of every error counting program P_w is decidable, even in the case of an extended version with recursive function calls.*

Every possible value range of the error counting variables is limited by the constant d_{min}. After all modifications of all error counting variables, the model checker evaluates the correctness assertions and returns a counterexample in the case of a violation. Therefore, in every program P_w no variable value ever exceeds $2 \cdot (d_{min} - 1)$. The domain of all variables is finite. Therefore, the resulting programs are effectively Boolean programs and the problem is reducible to solving a Boolean program. According to Ball and Rajamani [2], Boolean programs are equivalent to push-down automatons and therefore decidable [8].

7 Evaluation

The former sections described our method to verify the error masking robustness of encoded programs. Using this technique, we were able to identify real error masking vulnerabilities of real world, security relevant algorithms. Our set of algorithms under verification contains (among others) the following algorithms, which we want to describe in further detail: (1) Fibonacci Number Generator, (2) Euclidean Algorithm, (3) Extended Euclidean Algorithm, (4) Square & Multiply Exponentiation Algorithm and (5) Exponentiation in $\mathbb{Z}_n$. All of these iterative algorithms can be expressed in our toy language, with multiplication, division and modulo replaced by repeated addition and all function calls inlined. For further details on the algorithms, we refer to [19].

In our experiments, we used algorithms in the form of C source code, compiled them to LLVM bitcode, and generated the weight counting programs using a tool based on the LLVM compiler framework. Afterwards, we evaluated both a checkless version and a version containing correctly placed checks using the model checker CPAChecker [6]. Table 1 shows the verification time given different fault specifications. As configuration, we choose an iterative bounded model checking approach, where the loop bound is incremented if no error was found up to a limit of 5 loop iterations. This allowed us to calculate the exact loop bound where error masking occurs for the given specification. If the result is still unsound after a bounded model checking with an unroll bound of 5, we run a predicate analysis [5] algorithm to conclude the evaluation. Table 1 shows the verification time of the first algorithm with a sound result, on a machine with up to 16 threads running in parallel.

Table 1 shows that the complexity of the evaluation depends less on the number of injected bit flips, but more on the number of loop iterations necessary until error masking occurs, as well as the complexity (number and depth of nested loops) of P. Especially in the case of the last fault specification, d_{min} was greater than three times the maximum injectable error weight. In practise such a ratio and therefore this problem is quite unlikely, because a high d_{min} is costly (more redundant bits are necessary) and will not be chosen as protection against the injection of a way smaller number of bit flips.

Table 1. Verification times for different fault specifications.

d_{min}	FaultSpec	Program	Without checks			With correct checks	
			Ver. time	Iter.	Robust?	Ver. time	Robust?
2	$\sum W_i^v \leq 1$	(1) Fibonacci	1 s	2	✗	1 s	✓
		(2) Euclid	1 s	–	✓	–	–
		(3) Extended Euclid	8 s	2	✗	241 s	✓
		(4) Square & Multiply	16 s	2	✗	152 s	✓
		(5) Exp in $\mathbb{Z}_n$	53 s	2	✗	43 s	✓
20	$\sum W_i^v \leq 10$	(1) Fibonacci	1 s	2	✗	1 s	✓
		(2) Euclid	1 s	–	✓	–	–
		(3) Extended Euclid	11 s	2	✗	271 s	✓
		(4) Square & Multiply	11 s	2	✗	159 s	✓
		(5) Exp in $\mathbb{Z}_n$	48 s	2	✗	43 s	✓
300	$\sum W_i^v \leq 100$	(1) Fibonacci	1 s	3	✗	1 s	✓
		(2) Euclid	1 s	–	✓	–	–
		(3) Extended Euclid	70 s	3	✗	1497 s	✓
		(4) Square & Multiply	161 s	3	✗	547 s	✓
		(5) Exp in $\mathbb{Z}_n$	t/o 1800 s	?	?	28 s	✓
40	$\sum W_i^v \leq 10$	(1) Fibonacci	2 s	4	✗	1 s	✓
		(2) Euclid	1 s	–	✓	–	–
		(3) Extended Euclid	1528 s	4	✗	t/o (1800 s)	?
		(4) Square & Multiply	1043 s	3	✗	561 s	✓
		(5) Exp in $\mathbb{Z}_n$	t/o 1800 s	?	?	28 s	✓

Table 2. Comparison of evaluated programs.

Program	# Checks P	# Instr. P	# W_i^v in $P_{weights}$	# Instr. $P_{weights}$
(1) Fibonacci	1	70	12	219
(2) Euclid	0	68	11	186
(3) Extended Euclid	5	162	61	943
(4) Square & Multiply	2	136	51	765
(5) Exp in $\mathbb{Z}_n$	2	211	78	1126

Therefore, more iterations were necessary to detect error masking and the verification task was more difficult. More details about the programs under test can be found in Table 2.

As the results show, the complexity of the verification depends less on the number of injected bit flips, than on the complexity of the programs. The high number of bit flips is possible through abstracting the concrete variable values away and comes with advantages and drawbacks alike. The next section further discusses these challenges and gives ideas for future work.

8 Discussion and Future Work

Our technique to prove the absence of error masking brings along advantages but also holds potential for future work. Most important is the fact, that we evaluate abstraction of the original program. There are two main drawbacks of this: (1) Not every error with an arithmetic weight $\geq d_{min}$ automatically allows to form a new valid code word, this also depends on the actual encoded data. (2) Due to the discarded branch conditions, we might report spurious errors on infeasible paths through the program.

Nevertheless, there are important reasons and advantages of this decision: First, the abstraction gives us independence of the program argument's values. Therefore the search space for variable values is way smaller. Second, by storing the weights instead of the exact errors, the model checker does not need to calculate any arithmetic weight. This significantly reduces the complexity of the verification problem. Furthermore, the abstraction of the branch condition reduces the length of the path conditions and the algorithm *Predicate Analysis* solves the tasks independently of loop iterations. All these advantages help to decrease the verification effort.

However, this method just builds one step towards complete verification of robustness against injected faults. Both, the language and the fault model can be further extended. Including pointers and support for other encoding schemes (e.g. linear codes) may introduce new challenges and poses an interesting problem for the future.

9 Conclusion

In this article, we presented a novel method to verify the robustness against error masking of arithmetically encoded programs. This property guarantees that all faults according to the predefined fault model are detectable. The described technique applies formal methods to either prove the absence of error masking or calculate a counterexample. We provided a proof for the correctness of our approach and evaluated it using the model checker CPAChecker. Finally, a demonstration based on a real-world example multiplication algorithm shows the feasibility of our method.

References

1. Ali, S., Mukhopadhyay, D., Tunstall, M.: Differential fault analysis of AES: towards reaching its limits. J. Cryptogr. Eng. **3**, 73–97 (2013). https://doi.org/10.1007/s13389-012-0046-y
2. Ball, T., Rajamani, S.K.: Bebop: a symbolic model checker for boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000). https://doi.org/10.1007/10722468_7
3. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Proc. IEEE **94**, 370–382 (2006). https://doi.org/10.1109/JPROC.2005.862424

4. Baumann, R.C.: Radiation-induced soft errors in advanced semiconductor technologies. IEEE Trans. Device Mater. Reliab. **5**(3), 305–316 (2005)
5. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reason. **60**(3), 299–335 (2018)
6. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
7. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of eliminating errors in cryptographic computations. J. Cryptol. **14**, 101–119 (2001). https://doi.org/10.1007/s001450010016
8. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63141-0_10
9. Brown, D.T.: Error detecting and correcting binary codes for arithmetic operations. IRE Trans. Electron. Comput. **9**, 333–337 (1960). https://doi.org/10.1109/TEC.1960.5219855
10. Diamond, J.M.: Checking codes for digital computers. Proc. IRE **43**(4), 483–490 (1955). https://doi.org/10.1109/JRPROC.1955.277858
11. Fetzer, C., Schiffel, U., Süßkraut, M.: AN-encoding compiler: building safety-critical systems with commodity hardware. In: Buth, B., Rabe, G., Seyfarth, T. (eds.) SAFECOMP 2009. LNCS, vol. 5775, pp. 283–296. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04468-7_23
12. Given-Wilson, T., Heuser, A., Jafri, N., Lanet, J.L., Legay, A.: An automated and scalable formal process for detecting fault injection vulnerabilities in binaries (2017). https://hal.inria.fr/hal-01629135, working paper or preprint
13. Golay, M.: Notes on digital coding. Proc. IRE **37**(6), 657–657 (1949). https://doi.org/10.1109/JRPROC.1949.233620
14. Hamming, R.W.: Error detecting and error correcting codes. Bell Labs Tech. J. **29**(2), 147–160 (1950)
15. Kim, Y., et al.: Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. In: International Symposium on Computer Architecture – ISCA 2014, pp. 361–372 (2014)
16. Larsson, D., Hähnle, R.: Symbolic fault injection. In: Beckert, B. (ed.) Proceedings of 4th International Verification Workshop in connection with CADE-21. CEUR Workshop Proceedings, Bremen, Germany, 15–16 July 2007, vol. 259. CEUR-WS.org (2007). http://ceur-ws.org/Vol-259/paper09.pdf
17. Massey, J.L.: Survey of residue coding for arithmetic errors. Int. Comput. Cent. Bull. **3**(4), 3–17 (1964)
18. Medwed, M., Schmidt, J.-M.: Coding schemes for arithmetic and logic operations - how robust are they? In: Youm, H.Y., Yung, M. (eds.) WISA 2009. LNCS, vol. 5932, pp. 51–65. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10838-9_5
19. Menezes, A., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
20. Meola, M.L., Walker, D.: Faulty logic: reasoning about fault tolerant programs. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 468–487. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_25

21. Pattabiraman, K., Nakka, N., Kalbarczyk, Z.T., Iyer, R.K.: SymPLFIED: symbolic program-level fault injection and error detection framework. IEEE Trans. Comput. **62**(11), 2292–2307 (2013). https://doi.org/10.1109/TC.2012.219
22. Peterson, W.W.: Error-Correcting Codes. MIT Press, Cambridge (1961)
23. Rao, T.R.N.: Biresidue error-correcting codes for computer arithmetic. IEEE Trans. Comput. **19**(5), 398–402 (1970)
24. Rao, T.R.N., Garcia, O.N.: Cyclic and multiresidue codes for arithmetic operations. IEEE Trans. Inf. Theory **17**(1), 85–91 (1971)
25. Rink, N.A., Castrillón, J.: Extending a compiler backend for complete memory error detection. In: Dencker, P., Klenk, H., Keller, H.B., Plödereder, E. (eds.) Automotive - Safety and Security 2017 - Sicherheit und Zuverlässigkeit für automobile Informations technik. LNI, Stuttgart, Germany, 30–31 Mai 2017, vol. P-269, pp. 61–74. Gesellschaft für Informatik, Bonn (2017). https://dl.gi.de/20.500.12116/147
26. Schiffel, U.: Hardware error detection using AN-codes. Ph.D. thesis, Dresden University of Technology (2011). http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-69872
27. Schiffel, U.: Safety transformations: sound and complete? In: Bitsch, F., Guiochet, J., Kaâniche, M. (eds.) SAFECOMP 2013. LNCS, vol. 8153, pp. 190–201. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40793-2_18
28. Schilling, R., Werner, M., Mangard, S.: Securing conditional branches in the presence of fault attacks. In: Design, Automation and Test in Europe Conference and Exhibition – DATE 2018, pp. 1586–1591 (2018)
29. Selmke, B., Brummer, S., Heyszl, J., Sigl, G.: Precise laser fault injections into 90 nm and 45 nm SRAM-cells. In: Homma, N., Medwed, M. (eds.) CARDIS 2015. LNCS, vol. 9514, pp. 193–205. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-31271-2_12
30. Sharma, V.C., Haran, A., Rakamaric, Z., Gopalakrishnan, G.: Towards formal approaches to system resilience. In: IEEE 19th Pacific Rim International Symposium on Dependable Computing, PRDC 2013, Vancouver, BC, Canada, 2–4 December 2013, pp. 41–50. IEEE Computer Society (2013). https://doi.org/10.1109/PRDC.2013.14
31. Walker, D., Mackey, L.W., Ligatti, J., Reis, G.A., August, D.I.: Static typing for a faulty lambda calculus. In: Reppy, J.H., Lawall, J.L. (eds.) Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, 16–21 September 2006, pp. 38–49. ACM (2006). https://doi.org/10.1145/1159803.1159809
32. Werner, M., Unterluggauer, T., Schaffenrath, D., Mangard, S.: Sponge-based control-flow protection for IoT devices. CoRR abs/1802.06691 (2018). http://arxiv.org/abs/1802.06691