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
`firstname.lastname@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 errrors 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

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.

[⋆] 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 thank Karin Greiml for her support.

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. [7], [1]). 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 ([10], [22], [9]).

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`.

```
a := encode(0)
a := flip(a, 0th bit)
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.

```
a := encode(0)
a := flip(a, 0th bit)
b := a + a
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 0^{th} bit in `a` results in a flip of the 1^{st} 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.

```
a := encode(0)
a := flip(a, 0th bit)
b := a + a
check(b)
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 remaining sections are structured as follows: First, section 2 describes the state of the art and related work. Then, section 3 explains the concept of arithmetic codes as preliminaries and describes the most prominent examples. The main method is described in section 4 and section 5, where the former describes the input language and fault model, while the later describes the process to create an verifiable abstraction of the program under verification. section 6 proves the correctness of this approach and section 7 evaluates our method on a series of algorithms. Finally, section 8 provides an discussion of (dis-)advantages and an outline of future work, and section 9 summarizes this work.

2 Related Work

As section 1 illustrates, the detection of faults during program execution poses an important challenge. During the past decades, several interesting articles on this topic have been published.

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 identified 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 errors propagate 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 the first is to execute two versions of a program - the original and a faulted version - and to evaluate the user defined predicates. Every combination of program counter and 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 it 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 the complete 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 an 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 a 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 detected an error. The check macro for AN-codes is given in Listing 1.4.

Listing 1.4. Checks for AN encoded programs.

```
#define check(x) assert(x % A == 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 between the the two integer values x_1 and x_2 is equal to the arithmetic weight of the absolute difference between the values 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 of a program our method requires. It is defined as the minimum arithmetic distance between any two valid code words x_{c1} and x_{c2}. 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.

$$d_{min} = \min_{x_{c1} \neq x_{c2}} d(x_{c1}, x_{c2})$$

4 Error Masking Robust Programs

In this section, we describe the kind of programs we target and explain the concept of robustness against error masking. We define the error model and show a new method on how to apply this fault model to the program to transform it to a new program, where all potential faults are explicitly injected. Finally, we present a formal definition of robustness against error masking based on these explicitly faulted programs.

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, $\lambda : V \to S$ 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 $var \in Var$ are arithmetically encoded, and all constants $c \in \mathbb{N}$ are natural numbers. Values $val \in Var \cup Enc_c$ are either encoded variables or encoded constants $Enc_c = \{\texttt{encode}(c) \mid c \in \mathbb{N}\}$, where $\texttt{encode}(c)$ is the arithmetic encoded version of an integer constant c. All statements are either arithmetic instructions or control-flow directives $S = S_{arith} \cup S_{cf}$.

Arithmetic instructions can either be assignments $s \in S_{assign}$, additions $s \in S_{add}$, or subtractions $s \in S_{sub}$; $S_{arith} = S_{assign} \cup S_{add} \cup S_{sub}$.

$$\forall s \in S_{arith} :$$
$$s = \begin{cases} var := c_{enc} & \text{if } s \in S_{assign} \\ var := var_1 + var_2 & \text{if } s \in S_{add} \\ var := var_1 - var_2 & \text{if } s \in S_{sub} \end{cases}$$

Control-flow directives include direct jumps $s \in S_{jump}$, conditional branches $s \in S_{cbranch}$, runtime assertions $s \in S_{check}$ and terminators $s \in S_{ret}$; $S_{cf} = S_{jump} \cup S_{cbranch} \cup S_{check} \cup S_{ret}$.

$$\forall s \in S_{cf} :$$
$$s = \begin{cases} \texttt{goto } v & \text{if } s \in S_{jump} \\ \texttt{if } (cond) \texttt{ goto } v_1 \texttt{ else goto } v_2 & \text{if } s \in S_{cbranch} \\ \texttt{check } (var) & \text{if } s \in S_{check} \\ \texttt{return } var & \text{if } s \in S_{ret} \end{cases}$$

The runtime assertions $\texttt{check}\ (var)$ check a code word var for validity, abort execution, and enter a safe state once it detects any fault on this variable. However, checks cannot detect masked errors and only guarantee to disclose errors with a maximum arithmetic weight of $d_{min} - 1$. Their actual implementation depends on the encoding scheme of the program and is both possible in hardware or in software.

Boolean conditions $cond$ are either comparisons $val_i\ op\ val_j$, with $op \in \{<, \leq, =, \neq, \geq, >\}$ or boolean combinations of comparisons.

Every conditional branch performs an implicit check on its operands. To avoid the flipping the boolean value of $cond$ itself, we propose to use branch protection algorithms like [28]. The execution of a conditional branch can fall into one of three cases: One, every operand is correct, and the execution continues with $\texttt{goto}\ v_1$, if $cond$ evaluates to true or with $\texttt{goto}\ v_2$ otherwise. Second, 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. Third, the error weight on the compared operands exceed $d_{min} - 1$, where a branch protection mechanism could miss the fault. The statement continues with any of both `goto` statements and executes a possibly invalid path. The last case is a consequence of error masking and will be detected by our method.

Every vertex v_1 with a statement $\lambda(v_1) \in S_{arith} \cup S_{check} \cup S_{jump}$ has exactly one successor v_2. If $\lambda(v_1) =$ `goto` v_2, the destination vertex v_2 must be the single successor of v_1. For every vertex and its successor E contains a directed edge $(v_1, v_2) \in E$. All vertices v with conditional branch statements $\lambda(v) =$ `if` ↩ $(cond)$ `goto` v_2 `else goto` v_3 have exactly two outgoing edges to v_2 and v_3, and all vertices v with return statements $\lambda(v) \in S_{ret}$ have zero outgoing edges.

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.

As running example we use our small toy program from Listing 1.2 and Listing 1.3. The `flip` in this example was not intended by the programmer, it occurred due to either an attacker or environmental influences during execution. Listing 1.5 contains the original program, as it was written by the programmer.

Listing 1.5. Running example toy program.

```
myProgram()
  a := encode(0)
  b := a + a
  check(b)
  c := a + b
  return c
```

4.2 Fault Model

This work focuses on faults in memory, where bits of variable values are flipped. Every fault consists of an (possibly negative) error E of an arithmetic weight $W(|E|) < d_{min}$ added to an integer variable *var* at any point in time during program execution. A special case of faults are bit flips. A single bit flip in the i^{th} bit corresponds to an error $E = b_i 2^i$, 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 $W(|E|) = 1$. 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 such 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.

Definition 3 (Explicitly Faulted Program $P_{faulted}$).
Given a program $P = (V, E, \lambda, v_0, Var)$, we can derive an explicitly faulted program $P_{faulted} = (V_f, E_f, \lambda_f, v_{0_f}, Var_f)$, where $V_f = \{v' \mid v \in V\} \cup \{v'' \mid v \in V\}$, $E_f = E_{f_1} \cup E_{f_2}$ is a set of edges with $E_{f_1} = \{(v'', v') \mid v \in V\}$, $E_{f_2} = \{(v'_1, v''_2) \mid \exists v_2 \in V : (v_1, v_2) \in E\}$, $v_{0_f} = v'_0$, and $Var_f = \{var_f \mid var \in Var\}$.

An input program P is transformed into an explicitly faulted program $P_{faulted}$, $P \rightsquigarrow P_{faulted}$ by the following modifications. The new program $P_{faulted}$ is created by inserting an additional vertex v'_f before every vertex in a copy of V, $\{v''_f \mid v \in V\}$. Every statement of a vertex in the original program remains the same $\forall v \in V : \lambda_f(v''_f) = \lambda(v)[var \setminus var_f]$, with variables replaced by their explicitly faulted version. The statements on the new vertices $\lambda_f(v'_f)$ inject faults on every variable read by the statement $\lambda(v)$ of the original program.

$$\lambda_f(v'_f) = \begin{cases} var_f := var_f + E_1^v & \text{if} \quad \lambda(v) = \texttt{return}\ var_f \\ \begin{array}{l} var_{f_1} := var_{f_1} + E_1^v \\ var_{f_2} := var_{f_2} + E_2^v \end{array} & \text{if} \quad \begin{array}{l} \lambda(v) = var_{f_3} := var_{f_1} + var_{f_2} \vee \\ \lambda(v) = var_{f_3} := var_{f_1} - var_{f_2} \end{array} \\ \epsilon & \text{else} \end{cases}$$

In this formula, E_i^v denotes the error injected before execution the statement $\lambda(v)$ into the i^{th} operand of $\lambda(v)$.

The explicitly faulted version of our toy example is depicted in Listing 1.6.

Listing 1.6. $P_{faulted}$ of the toy example in Listing 1.5.

```
myProgram()
    a := encode(0)

    a := a + E_1^{v1}
    a := a + E_2^{v1}
    b := a + a

    check(b)

    a := a + E_1^{v2}
    b := b + E_2^{v2}
    c := a + b

    c := c + E_3^{v1}
    return c
```

4.4 Robustness Condition

The explicit faults in $P_{faulted}$ 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 $\pi = \pi[0], \ldots, \pi[k]$ is a sequence of vertices $\pi[i] \in V$, where the program graph P has a directed edge between any two subsequent elements $(\pi[j], \pi[j+1]) \in E$.*

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

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 also feasible in an explicitly faulted program $P_{faulted}$, 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, $\forall v \in V_f \forall i : E_i^v = 0$.*

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

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

Definition 10 (Correctness of an explicitly faulted program). *A faulted program P_f is correct if every feasible execution trace is also feasible in the fault-free program P_f^0, and all its executions return either the fault-free value $[\![\Pi[k] \mid var_f^0]\!]$ or any fault on the returned value $[\![\Pi[k] \mid var_f]\!]$ is detectable ($W(|E|) < d_{min}$).*

Definition 11 (Robustness of an input program). *A program P is robust against error masking, if and only if the explicitly faulted program $P_{faulted}$ is correct.*

To guarantee the robustness against error masking, the defined properties of Definition 10 are required to hold in 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 in the return value. Both problems are detected by the method described in the next section.

5 Proving a Program Robust against Fault Masking

This section describes, how to formally prove that a program fulfills the robustness condition described in section 4. The work flow of the verification of error masking robustness is depicted in Figure 1. Starting from a program P, we create the explicitly faulted program $P_{faulted}$ and derive an abstract model of the error weight propagation $P_{weights}$. This model contains assertions for the error masking robustness, and a model checker evaluates them for violations. In the case of a violation, the generated counterexample can be used to improve P, and insert additional checks. Once the model checker reports no errors any more, the program is guaranteed to be error masking robust.

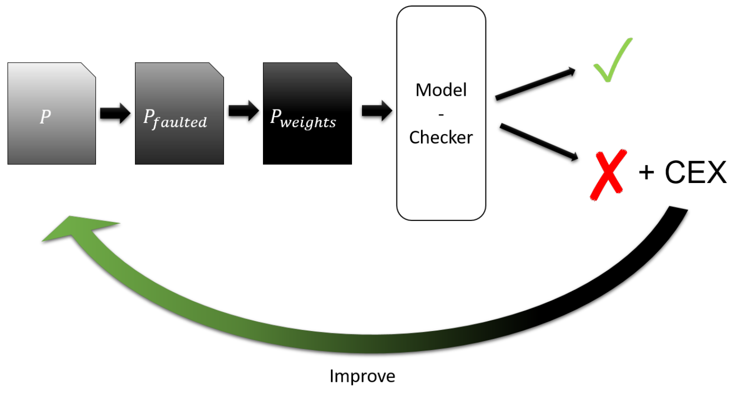

Fig. 1. The work flow of the verification process.

The main idea is to keep track of the maximum error weight on every variable and assert that it always remains below d_{min}. When this is the case, errors cannot mask each other and are always detectable.

Our technique to prove robustness involves three main steps: (1), we derive the explicitly faulted program $P \rightsquigarrow P_{faulted}$ from the input program P, as described in section 4. Afterwards, (2) transforms the faulted program $P_{faulted}$ into an error weight counting program $P_{weights}$. This program models the propagation of error weights through the explicitly faulted program $P_{faulted}$ and contains assertions for ensuring its correctness. (3), we apply an off-the-shelf model checker to evaluate the new program $P_{weights}$. The model checker proves the absence of error masking or provides a counterexample in case of any violations of the correctness assertions.

5.1 Adaption of the Input Language

The language of weight counting programs $P_{weights}$ is an adaption of the original language of P and P_f in subsection 4.1. Every statement $\lambda_w(v_w)$ on a vertex v_w is restricted to one of the following forms.

$$\lambda_w(v_w) \in \begin{cases} ew := * \\ ew := 0 \\ ew := ew_1 \\ ew := ew_1 + ew_2 \\ ew := ew + W_v^i \\ \texttt{goto } v' \\ \texttt{if } (*) \texttt{ goto } v_1 \texttt{ else goto } v_2 \\ \texttt{return} \\ \texttt{assume } (cond) \\ \texttt{assert } (ew < d_{min}) \end{cases}$$

In this syntax, the $*$ symbol denotes non-deterministic choice. The task of the model checker is to prove that for any value as $*$ the assertions $ew < d_{min}$ are never violated, given that all $cond$s provided to `assume` calls are fullfilled.

5.2 Fault Specification

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

Definition 12 (Maximum Injected Error Weight). *The maximum injected error weight W_i^v denotes the maximum number of errors injected over all visits to a vertex v in the i^{th} operand.*

Definition 13 (Fault Specification). *The fault specification FSpec is a Boolean expression over predicates E `op` c, with `op` $\in \{<, \leq, =, \geq, >, \neq\}$, E as a sum of maximum injected error weights W_i^v, and $c \in \mathbb{N}$, such that it restricts every injected error weight to an upper limit $< d_{min}$:*

$$\forall W_i^v : \ FSpec \implies W_i^v < d_{min}$$

.

A simple example for a fault specification would be to limit the sum of all maximum injected error weights to a constant $C < d_{min}$:

$$\sum W_i^v < C.$$

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

The error weight counting program $P_{weights}$ can be derived from an explicitly faulted program $P_{faulted}$, via the transformation $P_{faulted} \rightsquigarrow P_{weights}$. The error weight counting program is an abstraction of the program $P_{faulted}$, storing only the upper bound of the error weight on the corresponding variables' value. The set of error weight counters $WE = \{we \mid var_f \in Var_f\}$ is a set of unsigned variables $0 \leq we$.

As a start, the program $P_{weights}$ initializes ever error weight counter to 0. The vertex $v_w^{ew\_init}$ is the first vertex of the program with the following statements:

$$\lambda_w(v_w^{ew\_init}) = \{ew := 0 \mid var_f \in Var_f\}$$

Next, within the node v^{W_init}, every maximum injected error weight W_i^v is set to a non deterministic, positive integer:

$$\lambda_w(v_w^{W\_init}) = \{W_i^v := * \mid E_i^v\}$$

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

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

For every vertex $v_f \in V_f$ of $P_{faulted}$, there is a vertex $v_w \in V_w^{prog}$, with a statement $\lambda_w(v_w)$ as follows:

$$\forall \lambda_f(v_f) \in S_{arith}:$$
$$\lambda_w(v_w) = \begin{cases} ew := 0 & \text{if } \lambda_f(v_f) = var_f := c_{enc} \\ ew := ew_1 & \text{if } \lambda_f(v_f) = var_f := var_{f_1} + var_{f_1} \\ ew := ew_1 + ew_2 & \text{if } \lambda_f(v_f) = var_f := var_{f_1} + var_{f_2} \\ ew := ew_1 + ew_2 & \text{if } \lambda_f(v_f) = var_f := var_{f_1} - var_{f_2} \end{cases}$$

$$\forall \lambda_f(v_f) \in S_{cf}:$$
$$\lambda_w(v_w) = \begin{cases} \texttt{goto } v'_w & \text{if } \lambda_f(v_f) = \texttt{goto } v'_f \\ \begin{array}{l} \texttt{assert}(\bigwedge ew_i < d_{min}) \\ \texttt{assume}(\bigwedge ew_i = 0) \\ \texttt{if } (*) \texttt{ goto } v'_w \\ \texttt{else goto } v''_w \end{array} & \text{if } \lambda_f(v_f) = \begin{array}{l} \texttt{if } (cond \quad \texttt{goto } v'_f \\ \texttt{else goto } v''_f \end{array} \\ \begin{array}{l} \texttt{assert}(ew < d_{min}) \\ \texttt{assume}(ew = 0) \end{array} & \text{if } \lambda_f(v_f) = \texttt{check}(var_f) \\ \texttt{return} & \text{if } \lambda_f(v_f) = \texttt{return } var_f \end{cases}$$

$$\lambda_w(v_w) = \begin{array}{l} ew := ew + W_i^v \\ W_i^v := 0 \end{array} \qquad \text{if } \lambda_f(v_f) = var_f := var_f + E_i^v$$

In $P_{weights}$ 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 section 8. As all variables accessed by *cond* are implicitly checked by a branch protection algorithm (see subsection 4.1), the new statement begins with the transformed version of these `check`s.

Like in $P_{faulted}$, all fault injections are explicit. A fault injection in $P_{weights}$ 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.

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 $P_{weights}$, the correctness is assured by calls to the `assert` function.

$$\begin{aligned} V_w^{assert} = \{v_{err}^{assert} \mid \lambda_f(v_f') = var_f := var_f + E_i^{v_f}\} \\ \cup \{v_{arith}^{assert} \mid \lambda_f(v_f) = var_f := var_1 + var_2 \\ \vee \lambda_f(v_f) = var_f := var_1 - var_2\} \end{aligned}$$

$$\lambda_p(v^{assert}) = \texttt{assert}(ew < d_{min})$$

The assertion statement $\lambda_p(v_w^{assert})$ of each new vertex $v_w^{assert} \in V_w^{assert}$ checks that the error weight *ew* of the corresponding variable var_f is less than the minimum arithmetic distance.

In the resulting $P_{weights}$, the set of all vertices is $V_w = \{v_w^{ew\_init}, v_w^{W\_init}, v_w^{fs}\} \cup V_w^{prog} \cup V_w^{assert}$, with an edge between the initialization vertices $\{(v_w^{ew\_init}, v_w^{W\_init}), (v_w^{W\_init}, v_w^{fs})\} \in E_w$ and edges between the copied program vertices and the assertion vertices interleaved $\{(v_w^{P_f}, v_w^{assert}) \mid v_f \in V_f\} \cup \{(v_w^{assert}, v_w^{P_f}) \mid (v_f, v_f') \in E_f\}$. Finally, there is an edge from the last initialization vertex v_w^{fs} to the first vertex of V_w, which corresponds to v_{0_f}

Definition 14 (Weight Counting Program).
Given $P_{faulted} = (V_f, E_f, \lambda_f, v_{0_f}, Var_f)$*, we define*
$P_{weights} = (V_w, E_w, \lambda_w, v_{0_w}, Var_w)$.

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

The weight counting program of our toy example can be seen in Listing 1.7.

Listing 1.7. Weight counting version of the toy example.

```
myProgram()
    $W_1^{v_1}$, $W_1^{v_2}$, $W_2^{v_1}$, $W_2^{v_2}$, $W_3^{v_1}$ := *
    ew_a, ew_b, ew_c := 0
    assume($\sum_{v \in V} \sum_{i=1}^{\infty} W_i^v \leq 1$)
    ew_a := 0

    ew_a := ew_a + $W_1^{v_1}$
    assert(ew_a < $d_{min}$)
    ew_a := ew_a + $W_2^{v_1}$
    assert(ew_a < $d_{min}$)
    ew_b := ew_a
    assert(ew_b < $d_{min}$)

    assert(ew_b < $d_{min}$)
    assume(ew_b = 0)

    ew_a := ew_a + $W_1^{v_2}$
    assert(ew_a < $d_{min}$)
    ew_b := ew_b + $W_2^{v_2}$
    assert(ew_b < $d_{min}$)
    ew_c := ew_a + ew_b
    assert(ew_c < $d_{min}$)

    ew_c := ew_c + $W_3^{v_1}$
    assert(ew_c < $d_{min}$)
    return
```

5.4 Applying a Model Checker to Prove Correctness

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

The next section will give an 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, $P_{weights}$ contains an assertion violation. For this, we use the following definitions.

Definition 15 (Mapping of a Program State). *Given a program state of the explicitly faulted program $P_{faulted}$, $\Pi_f[t]$, we define $\Pi_w(\Pi_f[t])$ as the corresponding program state of $P_{weights}$, where $\forall E_i^v : W(|[\![\Pi_f[0] \mid E_i^v]\!]|) = [\![\Pi_w(\Pi_f[0]) \mid W_i^v]\!]$, and $[\![\Pi_w(\Pi_f[t])]\!]_\pi$ is the smallest execution trace containing $[\![\Pi_f[t]]\!]_\pi$.*

Theorem 1. *Given a program state $\Pi_f[t]$, where every variable is smaller or equal to its corresponding error weight counter in $\Pi_w(\Pi_f[t])$, 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 the error weight counter ew belonging to this variable.*

Proof. All arithmetic statements fall into one of the following cases:

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.

$$W(|E(\texttt{encode}(c))|) = 0 \implies ew = 0 \geq W(|E(var_f)|).$$

For the addition of the same variable with itself, $\lambda_f(v_f) = var_f := var_{f_1} + var_{f_1}$, we get

$$D(var_{f_1} + var_{f_1}, var^0_{f_1} + var^0_{f_1}) = W(|2E(var_{f_1})|) = W(|E(var_{f_1})|)$$
$$\implies ew = W(|E(var_{f_1})|) = W(|E(var_f)|).$$

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:

$$\begin{aligned} D(var_{f_1} + var_{f_2}, var^0_{f_1} + var^0_{f_2}) &= W(|E(var_{f_1}) - E(var_{f_2})|) \\ &\leq W(|E(var_{f_1})|) + W(|E(var_{f_2})|) \\ \implies ew = W(|E(var_{f_1})|) + W(|E(var_{f_2})|) &\geq W(|E(var_f)|). \end{aligned}$$

Theorem 2. *In any program state $\Pi_f[t]$ of $P_{faulted}$ with $\Pi_w(\Pi_f[t])$ fulfilling all assumed conditions, the error of a variable $[\![\Pi_f[t] \mid E(var_f)]\!]$ has at most the arithmetic weight stored in the corresponding error weight variable ew, $[\![\Pi_f[t] \mid E(var_f)]\!] \leq [\![\Pi_w(\Pi_f[t]) \mid ew]\!]$.*

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; Therefor it is correct to assume that all variable's error weight are 0.

Suppose all error weights in every program state $\Pi_w(\Pi_f[i])$ with $i < t$ are correct. $\forall i < t. \forall var_f [\![\Pi_f[i] \mid E(var_f)]\!] \leq [\![\Pi_w(\Pi_f[i]) \mid ew]\!]$. 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 $[\![\Pi_f[t+1] \mid E(var_f)]\!] \leq [\![\Pi_w(\Pi_f[t+1]) \mid ew]\!]$:

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 $\forall t \Pi_f[t] = w(\Pi_f[t])$, the control-flow directives will not influence any error weights. Finally, given Definition 15 defines that all $\forall E^v_i : W(|[\![\Pi_f[0] \mid E^v_i]\!]|) = [\![\Pi_w(\Pi_f[0]) \mid W^v_i]\!]$. This guarantees that $[\![\Pi_f[t+1] \mid E(var_f)]\!] \leq [\![\Pi_w(\Pi_f[t+1]) \mid ew]\!]$.

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 an violation of the assertion $\texttt{assert}(ew < d_{min})$, or that $E(var_f) = 0$.*

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

1. $0 < W(|E(var_f)|) < d_{min}$ In this case, the check is not passed and the execution is aborted. No further error masking can occur.
2. $W(|E(var_f)|) \geq d_{min}$ If the error weight exceeds the minimum arithmetic distance, Theorem 2 proves that $ew \geq W(|E(var_f)|)$, and the assertion `assert`$(ew < d_{min})$ is violated.
3. $var_f = 0$ 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 $P_{weights}$ containing loops, where all error weights are injected in the first iteration, and a program $P'_{weights}$ abstracting the same program P, with all error weight injections distributed over all infinite loop iterations, it is always true that if $P_{weights}$ is correct, then $P'_{weights}$ 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. $[\![\Pi_w[t] \mid ew]\!] = \sum_{j=0}^{\infty} k_i^v[j] W_i^v[j]$, where the factor k_i^v indicates the number of times the injected error weight has accumulated in an error weight counter, and $W_i^v[j]$ is the error weight injected in loop iteration j. In the case of $P_{weights}$, $W_i^v[0] = W_i^v$ and $\forall j > 0 : W_i^v[j] = 0$, while all $W_i^v[j]$ of $P'_{weights}$ are smaller or equal those of $P_{weights}$. Furthermore, $\forall j > 0 : k_i^v[0] = 0 \vee k_i^v[0] > k_i^v[j]$, therefore, the only way that $[\![\Pi_w[t] \mid ew]\!] < [\![\Pi'_w[t] \mid ew]\!]$ can be achieved is, if ew is overwritten after injecting $W_i^v[0]$ ($k_i^v[0] = 0$), and j is the current loop iteration. However, in the next loop iteration, this error weight will be overwritten again ($k_i^v[j] = 0$). The maximum value during the first loop iteration will never be exceeded.

Theorem 5 (Correctness of $P_{weights}$). *If $P_{weights}$ is correct, P_{faults} is correct and P is robust against error masking.*

Proof. Assume $P_{faulted}$ is incorrect. Let $\Pi_f[k]$ be the last execution state of a program run violating the correctness of $P_{faulted}$, 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 $[\![\Pi_f[k] \mid W(|E(var_{ret})|)]\!] \geq d_{min}$, or (2), an invalid path through the program is taken.

In case (1), Theorem 2 provides a proof, that $[\![\Pi_f[k], E_{var_{ret}}]\!] > d_{min} \implies [\![\Pi_w(\Pi_f[k]) \mid ew_{ret}]\!]$. Therefore, at least the last assertion in $P_{weights}$ is violated and $P_{weights}$ 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 $P_{weights}$ detect this case as well, and therefore $P_{weights}$ is incorrect in this case, too.

This shows, that any violation of $P_{faulted}$ will always result in a violation of $P_{weights}$, and $P_{weights}$ correct $\implies$ $P_{faulted}$ correct.

Theorem 6 (Decidability). *The correctness of every error counting program $P_{weights}$ 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 $P_{weights}$ 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 check-less 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.

As Table 1 shows, 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 choosen as protection against the injection of a way smaller number of bit flips.

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.

			without checks			with correct checks	
d_{min}	FaultSpec	Program	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 1. Verification Times for Different Fault Specifications.

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

Table 2. Comparison of Evaluated Programs.

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. Cryptographic Engineering **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: SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings. pp. 113–130 (2000)
3. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Proceedings of the 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 Transactions on Device and materials reliability **5**(3), 305–316 (2005)
5. Beyer, D., Dangl, M., Wendler, P.: A unifying view on smt-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018)
6. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: CAV. Lecture Notes in Computer Science, vol. 6806, pp. 184–190. Springer (2011)
7. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of eliminating errors in cryptographic computations. J. Cryptology **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: CONCUR '97: Concurrency Theory, 8th International Conference, Warsaw, Poland, July 1-4, 1997, Proceedings. pp. 135–150 (1997)
9. Brown, D.T.: Error detecting and correcting binary codes for arithmetic operations. IRE Trans. Electronic Computers **9**, 333–337 (1960). https://doi.org/10.1109/TEC.1960.5219855
10. DIAMOND, J.M.: Checking codes for digital computers. Proceedings of the IRE **43**(4), 483–490 (April 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.) Computer Safety, Reliability, and Security, 28th International Conference, SAFECOMP 2009, Hamburg, Germany, September 15-18, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5775, pp. 283–296. Springer (2009). https://doi.org/10.1007/978-3-642-04468-7_23, `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 (Nov 2017), `https://hal.inria.fr/hal-01629135`, working paper or preprint
13. Golay, M.: Notes on digital coding. Proceedings of the IRE **37**(6), 657–657 (June 1949). https://doi.org/10.1109/JRPROC.1949.233620
14. Hamming, R.W.: Error detecting and error correcting codes. Bell Labs Technical Journal **29**(2), 147–160 (1950)
15. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: 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, Bremen, Germany, July 15-16, 2007. CEUR Workshop Proceedings, 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. International Computation Center Bulletin **3**(4), 3–17 (1964)
18. Medwed, M., Schmidt, J.: Coding schemes for arithmetic and logic operations - how robust are they? In: Youm, H.Y., Yung, M. (eds.) Information Security Applications, 10th International Workshop, WISA 2009, Busan, Korea, August 25-27, 2009, Revised Selected Papers. Lecture Notes in Computer Science, vol. 5932, pp. 51–65. Springer (2009). https://doi.org/10.1007/978-3-642-10838-9_5, `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 (1996)
20. Meola, M.L., Walker, D.: Faulty logic: Reasoning about fault tolerant programs. In: Gordon, A.D. (ed.) Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6012, pp. 468–487. Springer (2010). https://doi.org/10.1007/978-3-642-11957-6_25, `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. Computers **62**(11), 2292–2307 (2013). https://doi.org/10.1109/TC.2012.219, `https://doi.org/10.1109/TC.2012.219`
22. Peterson, W.W.: Error-correcting codes. M.I.T. Press [u.a.] (1961)
23. Rao, T.R.N.: Biresidue error-correcting codes for computer arithmetic. IEEE Trans. Computers **19**(5), 398–402 (1970)
24. Rao, T.R.N., Garcia, O.N.: Cyclic and multiresidue codes for arithmetic operations. IEEE Trans. Information 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 & Security 2017 - Sicherheit und Zuverlässigkeit für automobile Informationstechnik, Stuttgart, Germany, Mai 30-31, 2017. LNI, 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.) Computer Safety, Reliability, and Security - 32nd International Conference, SAFECOMP 2013, Toulouse, France, September 24-27, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8153, pp. 190–201. Springer (2013). https://doi.org/10.1007/978-3-642-40793-2_18, `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 & Test in Europe Conference & 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: Smart Card Research and Advanced Applications – CARDIS 2015. pp. 193–205 (2015)

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, December 2-4, 2013. pp. 41–50. IEEE Computer Society (2013). https://doi.org/10.1109/PRDC.2013.14, `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, September 16-21, 2006. pp. 38–49. ACM (2006). https://doi.org/10.1145/1159803.1159809, `http://doi.acm.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`