# Placement of Runtime Checks to Counteract Fault Injections

Benedikt Maderbacher[0000−0002−5834−352X], Anja F. Karl[000−0002−3062−7459], and Roderick Bloem[0000−0002−1411−5744]

Graz University of Technology, Graz, Austria
firstname.lastname@iaik.tugraz.at

**Abstract.** Bitflips form an increasingly serious problem for the correctness and security of software and hardware, whether they occur inadvertently as soft errors or on purpose as fault injections. Error Detection Codes add redundancy and make it possible to check for faults during runtime, making systems more resilient to bitflips. Codes require data integrity to be checked regularly. Such checks need to be used sparingly, because they cause runtime overhead.

In this paper, we show how to use static verification to minimize the number of runtime checks in encoded programs. We focus on loops, because this is where it is important to avoid unnecessary checks. We introduce three types of abstractions to decide correctness: depending on (i) whether we keep track of errors precisely or of their Hamming weights, (ii) how we check whether faults can still be detected, and (iii) whether we keep track of the data or not. We show that checks in loops induce simple and natural loop invariants that we can use to speed up the verification process.

The abstractions let us trade verification time against the number of required runtime checks, allowing us to find efficient sets of integrity checks for critical program fragments in reasonable time. Preliminary experimental data shows that we can reduce the number of runtime checks by up to a factor of ten.

## 1 Introduction

Fault injection attacks and soft errors [BBKN12] are a significant and growing concern in software security. By flipping bits, an attacker can reveal cryptographic secrets, change branching decisions, circumvent privilege evaluations, produce system failures, or manipulate the outcome of calculations. For instance, Boneh et al. [BDL97] showed how to break several implementations of cryptographic algorithms by injecting register faults. To prevent such attacks, they propose to protect the data integrity with error detection measures. Such countermeasures have been studied in detail for cryptographic algorithms [LRT12] [SFES18] [MAN$^+$18], but not as much for secure software in general, where fault attacks may also be problematic [YSW18]. Similarly, radiation can lead to random bit flips known as *soft errors* or *single event upsets*. Such errors were initially only problematic for hardware used in space, but with decreasing feature sizes, they have also become relevant for consumer electronics [MER05].

We address error detection with Error Detecting Codes (EDCs) [Ham50,Dia55]. The fundamental principle of EDCs is an injective mapping of *data words* to *code words* in an encoded domain. The mapping cannot be surjective, indeed, if a code word is valid (the image of a data word), then flipping a few bits should not yield another valid code word. Thus, limited faults can be detected using runtime checks.

We are interested in error detecting codes that are homomorphic over certain operations. For example, arithmetic codes preserve (some) arithmetic operations, and binary linear codes preserve bitwise logical operations. Thus, we can execute certain programs in an encoded domain, without the need to decode and reencode in-between operations.

We define the distance between code words as their Hamming distance (we will be more precise below). For a given code, the minimal distance between two valid code words is called the *distance of the code*, denoted by $d_{min}$. The maximal number of bitflips that a code can detect is thus $d_{min} - 1$. In a program, bitflips may propagate and grow. For instance, if a variable `a` contains one bitflip, `3 * a` may contain two.

We need to ensure that errors are not *masked*, which happens if combining two invalid code words results in a valid code word. We prevent masking by inserting *runtime checks* in the program that halt the program if the variable passed to the check is not a valid code word. Because checks cost runtime, we want to insert as few checks as possible.

In this paper, we show how to use static reasoning to minimize the number of runtime checks in encoded programs, building on the formal verification techniques presented in [KSBM19]. We pay special attention to the verification of checks inside loops, as their impact on correctness and on verification complexity is especially high. In order to help minimize the placement of checks, we make two contributions.

As a first contribution, we introduce a refined abstraction scheme. Where [KSBM19] only tracks the weight of the errors, we introduce a four-tiered abstraction scheme in which an error is tracked either by weight or by precise value; checks are performed either by checking the weight of error or by checking that a fault is not a valid code word itself, and finally the actual values of the variables are either abstracted away or kept precisely.

The second contribution is based on the observation that checks that are placed in loops induce simple loop invariants. We thus propose invariants that along with generated checks are used to reduce the verification of the runtime checks in the program to straight-line code.

We show experimentally that our approach allows us to significantly minimize the number of necessary runtime checks, by up to a factor of ten. The different levels of abstraction allow us to trade off the number of runtime checks in the program (we can prove programs with fewer checks correct if we have a finer abstraction) against the scalability of the approach (which is better if we abstract more strongly). The resulting approach allows us to insert efficient sets of checks into programs with reasonable computational effort.

## 2 Preliminaries

### 2.1 Fundamentals of Error Detecting Codes

Before introducing specific arithmetic codes, let us introduce some theory that is common across these types of codes. The fundamental operating principle of EDCs is the extension of every value with additional, redundant information. The key component of each EDC is the *encode* function. This function defines how a *data word* $w$ of the decoded domain $\mathcal{W}$, $w \in \mathcal{W}$, is mapped to a *code word* $c$ in the encoded domain $\mathcal{C}$:

$$\texttt{encode} : \mathcal{W} \mapsto \mathcal{C}.$$

Typically, both domains are an interval of non-negative integers. The function `encode` should be injective but not surjective, so that we can decode code words and detect (some) corruptions of code words. We call a code word $c$ *valid*, if it is part of the image of `encode`, i.e., $\exists\, w \in \mathcal{W} \;:\; c = \texttt{encode}(w)$, and *invalid* otherwise.

For the error detecting codes that are of interest here, we can define a distance function $d : \mathcal{C} \times \mathcal{C} \rightharpoonup \mathbb{N}$ on the encoded domain. (Distance functions fulfill the usual properties of non-negativity, identity, symmetry, and the triangle inequality.) We use these distances to measure faults (as distances to the intended value) and we will see below that they allow us to track these faults through certain types of operations.

The error detection capabilities of an error detecting code are limited by the minimum distance $d_{min}$ between any two valid code words [Ham50]. We define $d_{min}$ as the minimum distance between any two valid code words, i.e., $d_{min} = \min_{c_\mathrm{v}, c'_\mathrm{v}} d(c_\mathrm{v}, c'_\mathrm{v})$. We also define a weight function $\mathrm{weight} : \mathcal{C} \mapsto \mathbb{N}$ as the distance between a code word and the identity $\texttt{encode}(0)$, i.e., $\mathrm{weight}(c) = d(c, \texttt{encode}(0))$. Finally, we define a partial function $\texttt{decode} : \mathcal{C} \rightharpoonup \mathcal{W}$, as the inverse of the encode function, and a function $\texttt{isvalid} : \mathcal{C} \mapsto \mathbb{B}$, which maps valid code words to `true` and invalid code words to `false`. The result of `decode` is only defined for valid code words.

Most EDCs are homomorphic over a set of supported operations $\circ$, i.e.,

$$\texttt{encode}(w_1 \circ w_2) = \texttt{encode}(w_1) \circ \texttt{encode}(w_2),$$

which allows us to encode a whole program and execute the calculation directly in the encoded domain.

### 2.2 Arithmetic Codes

Our work targets programs protected by codes that are homomorphic over arithmetic operations.

*Arithmetic Codes* are homomorphic over operations like addition. The distance and weight functions are defined as the (base-2) *arithmetic distance* and *weight*, resp. [Mas64]:

$$\text{weight}_{\text{arit}}(c) = \min\bigl\{n \mid c = \sum_{i=0}^{n} a_i \cdot 2^{k_i} \text{ for some } a_i \in \{-1,1\} \text{ and } k_i \geq 0\bigr\}$$

and

$$d_{\text{arit}}(c_1, c_2) = \text{weight}_{\text{arit}}(|c_2 - c_1|).$$

For instance, $d_{\text{arit}}(9,2) = \text{weight}_{\text{arit}}(7) = 2$, because $7 = 8 - 1$. The distance between a value $c$ and a value $c'$ that differ only in one bit is one.

For any linear operation $\circ$, we have the propagation laws $d_{\text{arit}}(c_1 \circ c_2, c_1' \circ c_2') \leq d_{\text{arit}}(c_1, c_1') + d_{\text{arit}}(c_2, c_2')$ and $d_{\text{arit}}(c \circ c, c' \circ c') \leq d_{\text{arit}}(c, c')$.

*Example 1.* Separate multiresidue codes. Separate multiresidue codes [Rao70,RG71] are arithmetic codes. Every code word is a $(k+1)$-tuple and operations are performed separately on each element [Gar66,Pet58]. Every code is defined by $k$ constants, $m_1, \ldots, m_k$ and we encode a data word as

$$\texttt{encode}(w) = (w, |w|_{m_1}, \ldots, |w|_{m_k}),$$

where each check digit $|w|_{m_i}$ equals $w \mod m_i$. Every operation on the check digits is performed modulo its check base, so that

$$|w_1 \circ w_2|_{m_i} = ||w_1|_{m_i} \circ |w_2|_{m_i}|_{m_i},$$

making separate multiresidue codes are homomorphic over these operations. The constant $d_{min}$ depends on the choice of the check bases $m_1, \ldots m_k$ [MS09].

*Example 2.* AN-codes. For AN-codes, we fix a constant $A$ and we define $\texttt{encode}(w) = A \cdot w$. A code word is valid if its residue after division by $A$ is zero. Note that multiplication by $A$ distributes over addition and subtraction. The $d_{min}$ of the code does not follow by easy inspection of $A$, but we often choose $A$ to be a prime. (Note that a power of two would be a particularly poor choice with $d_{min} = 1$.)

## 2.3 Error Propagation

When considering computations, we consider both the *correct* value $c^0$ of a variable $c$, which occurs in an execution in which no faults are introduced, and its possible *faulty* counterpart $c^*$, which occurs in the corresponding execution where faults have been introduced. The *error weight* $e_c$ is defined as the distance $d(c^*, c^0)$ between the correct and actual code values. Recall that if $e_c \geq d_{min}$, then $c^*$ may equal the encoding of a different data word, and it may no longer be possible to detect a fault. On the other hand, if we can guarantee that $e_c < d_{min}$ for all variables $c$, then faults will remain detectable.

We will assume that faults are introduced as one or more individual bitflips. If a single bitflip is introduced in an otherwise correct variable $c^*$, we have that

**Table 1.** Error propagation rules for arithmetic codes. The symbol $\pm$ stands for addition or subtraction.

$$
\begin{aligned}
e_{v \pm v'} &\leq e_v + e_{v'} \\
e_{v \pm v} &\leq e_v \\
e_{-v} &= e_v
\end{aligned}
$$

$d(c^0, c^*) = 1$ and a bitflip introduced in an existing variable increases the error weight by at most one. By the propagation laws stated above, for any operation $\circ$, we have that $e_{c \circ c'} \leq e_c + e_{c'}$. Thus, faults may spread across an execution and "grow up": a variable may eventually have an error weight that is larger than the total number of bitflips introduced into the program. For example: $e_{(c \circ c') \circ c} \leq 2 * e_c + e_{c'}$ which can be larger than the sum of the injected errors $e_c + e_{c'}$.

Table 1 summarizes the propagation rules, where we use arithmetic error weights.

## 2.4 Fault Model

Our approach is relatively independent of the precise fault model chosen, as long as it can be modeled by a program transformation.

We illustrate our approach using a simple fault model in which faults consist of bit flips on memory variables. We model faults to be transient in the sense that the occurrence of a failure at a given program location does not mean the failure will occur again when the program location is visited again. We do, however, assume that errors persist in the sense that when the value of a variable has changed, it does not automatically change back again. This models faults in memory rather that faults on, say, the data path of a processor.

To model faults, we assume a minimal programming language with assignments, arithmetic operations, jumps, conditionals, and a runtime check statement. When called on a code word $c$, `check(c)` halts the program when $c$ is not a valid code word and continues otherwise. Note that this check will not detect code words that contain such a large error that they equal a different code word.

We model faults by a simple program transformation in which a program $P$ is transformed to an *annotated program* $P_f$. In $P_f$, we add a new statement `flip(v)` before every use of a variable `v`. This statement may nondeterministically flip one bit of the value of `v`. We will also typically have an *error assumption* $\varphi_\varepsilon$ that limits the number of bit flips that can be inserted into a run of the program. An example would be a specification that says that the total number of inserted bit flips is smaller than $d_{min}$. We refer to [KSBM19] for a formalization of this approach.

We assume that the control flow of the program is protected using other means [SWM18,WUSM18].

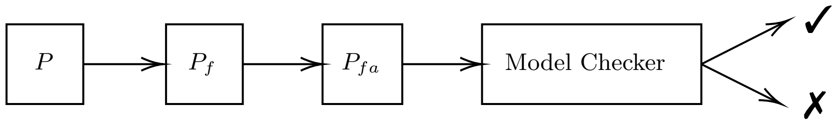

**Fig. 1.** Overview of the verification algorithm.

## 3 Error Tracking

In this section and the next, we describe how we verify the correctness of the checks in a program. We will thus define a code transformation from a program with loops into a set of loop-free programs with assumptions and assertions such that the correctness of the original program under the fault model is guaranteed if all of the loop-free code fragments are correct. This is done by tracking the the values of the arithmetic errors in addition to the values of the variables. This section introduces increasingly precise abstractions for error variables, to make the error tracking feasible. In the next section, we will describe how to generate the invariants necessary to handle loops.

Figure 1 contains a schematic of our verification approach. Starting with an encoded program $P$ in the first step the fault model is made explicit resulting in a program $P_f$. In the next step we apply one of the error abstractions described in this section, abstract the control flow and add assertions. This program $P_{fa}$ is then given to a model checker, which gives a verdict whether the program is secure.

The idea of the abstraction is to track the errors in the variables separately from their (intended, uncorrupted) values. In order to do this, we need the following property.

**Definition 1.** *An error-correcting code is error-homomorphic for a set $F$ of operations, if for any $f \in F$ there is an $f'$ such that*

$$f(a + \varepsilon_a) = f(a) + f'(\varepsilon_a) \text{ and}$$
$$f(a + \varepsilon_a, b + \varepsilon_b) = f(a, b) + f'(\varepsilon_a, \varepsilon_b),$$

*where + denotes addition, as we are dealing with arithmetic codes.*

AN-codes and multiresidual codes are error-homomorphic for addition and multiplication with constants. Multiresidual codes are error-homomorphic for multiplication as well, but AN-codes are not.

In effect, these constraints state that we can track the values of the variables separately from the errors. This is important for verification, because it means we can distinguish between three situations: (1) a value may be *correct*, that is, the error is zero. We denote this by corr($\varepsilon$); (2) the error may be *detectable* in a given code, denoted by detect($\varepsilon$); (3) the error may be *masked*, meaning that it is not zero but cannot be detected using the given code, denoted by masked($\varepsilon$). The third case is the one we want to avoid.

*Example 3.* For an AN code with constant $A$, $\text{corr}(\varepsilon)$ is defined as $\varepsilon = 0$, $\text{detect}(\varepsilon) = (\varepsilon \mod A \neq 0)$ and $\text{masked}(\varepsilon) = (\varepsilon > 0 \wedge \varepsilon \mod A = 0)$

We will construct a program $P_{fa}$ from $P_f$. For every variable `v` in $P_f$, $P_{fa}$ will have two variables, `v` and $\varepsilon_{\texttt{v}}$. We distinguish our abstractions along three dimensions:

1. The first question is how to keep track of errors that are introduced. We can track the actual arithmetic error, or we can abstract it away by keeping track of the weight of the error only.
2. We can vary how we check whether the induced errors can be handled by the given code: we can either check whether the concrete error can be detected by the given code, or we can abstract this to a check whether the weight of the fault is greater than $d_{min}$.
3. Finally, we can keep the actual values of the variables or we can fully abstract these away.

The abstractions are modeled as follows. In the following, we will assume static single assignment form and we will introduce assumptions on relations between variables when we cannot use assignments.

1. If we keep track of errors by their actual values, then for every `v`, $\varepsilon_{\texttt{v}}$ is a bitvector that models the error and the statement `flip(v)` is replaced by a statement $\texttt{flip}^a(\varepsilon_{\texttt{v}})$ that nondeterministically flips one of the bits in $\varepsilon_{\texttt{v}}$. We replace an assignment `u := f(v,w)` in $P_f$ by the two statements `u := f(v,w);` $\varepsilon_{\texttt{u}}$ `:=` $\texttt{f}'(\varepsilon_{\texttt{v}},\varepsilon_{\texttt{w}})$, using error homomorphism.
   If we keep only the weights of the errors, then $\varepsilon_{\texttt{v}}$ is a positive number, and $\texttt{flip}^a(\varepsilon_{\texttt{v}})$ nondeterministically adds one to $\varepsilon_{\texttt{v}}$. In these cases we replace an assignment `u := f(v,w)` by the statements `u := f(v,w); assume(`$\varepsilon_{\texttt{u}} \leq \texttt{f}'(\varepsilon_{\texttt{v}},\varepsilon_{\texttt{w}})$`)`. The value of $\varepsilon_{\texttt{u}}$ can be anything satisfying the assumption. Note that for arithmetic codes and additions, for instance, the weight of the error of the sum of two variables can be smaller than the sum of the weights of the errors.
2. A check whether the weight of a given error is greater than $d_{min}$ is easily implemented, whether or not we keep track of the concrete value of the error or only of its weight. If we keep track of the concrete value of an error, we can check make sure that the error value can be detected. For multiresidual codes with constants $m_1, \ldots, m_k$, this is the case if $\varepsilon_{\texttt{v}}$ is not a common multiple of $m_1, \ldots, m_k$; for AN-codes it is the case if it is not a multiple of $A$.
3. Finally, if we abstract away the values of the concrete variable of the program, we simply remove all assignments to the variables and replace conditionals with nondeterministic choices.

Based on these three dimensions, we define four levels of abstraction, as sketched in Table 2.

The abstract program is extended by adding assumptions and assertions. Whenever a variable gets assigned it may not contain error masking. The error must be either zero or detectable by the code.

**Table 2.** Abstraction Levels

| Level | Errors | checks | values |
|---|---|---|---|
| 3 | weight | weight | abstract |
| 2 | precise | weight | abstract |
| 1 | precise | code word | abstract |
| 0 | precise | code word | precise |

– For only checking the weight of the variables the assertion for a variable `v` is `assert(`$\mathrm{weight}(\varepsilon_{\mathtt{v}}) < d_{min}$`)`.
– If we check the actual code words the assertion instead is `assert(`$\mathrm{corr}(\varepsilon_{\mathtt{v}}) \vee \mathrm{detect}(\varepsilon_{\mathtt{v}})$`)`.

After a check on variable we added an assumption that the error on this variable is zero, as the program would abort otherwise. We can slightly reduce the number of assertions by only checking the variables when they are checked or used in some form of output.

**Theorem 1.** *If no assertion in the abstract program $P_{fa}$ is violated then either the program $P_f$ with faults conforming to the fault model $\varphi_\varepsilon$ raises an error or the output of $P$ and $P_f$ is equal.*

We overapproximate the control flow and the propagation of errors. Thus if no assertion is violated we can guarantee that no fault can lead to error masking and no manipulated values are in the program output. The other direction, however, is not true. There are programs that are rejected by our approach, that are secure against the fault model.

*Example 4.* The different behaviors of these abstraction levels can be demonstrated with a simple example. Let the following program, $P$, be protected by an AN code with $A = 7$, for which we have $d_{min} = 2$:

```
m := m + a;
check(m); check(a);
```

For the sake of a simpler presentation we only consider one error injection location on the variable `a` at the beginning of the program, so that our annotated program $P_f$ becomes

```
flip(a);
m := m + a;
check(m); check(a);
```

Let us assume that at most one bit is flipped. Using Abstraction Level 3 we obtain an abstract program $P_{fa}$. Combining that with the specification that puts the error at zero at the beginning at the program and requires safe errors at the end, we get the following.

```
assume($\varepsilon_{\mathtt{a}} = 0 \wedge \varepsilon_{\mathtt{m}} = 0$);
$\varepsilon'_{\mathtt{a}}$ := flip$^a$($\varepsilon_{\mathtt{a}}$);                // $\varepsilon'_{\mathtt{a}} = 1,\ \varepsilon_{\mathtt{m}} = 0$
assume($\varepsilon'_{\mathtt{m}} \leq \varepsilon_{\mathtt{m}} + \varepsilon'_{\mathtt{a}}$);            // $\varepsilon'_{\mathtt{a}} = 1,\ \varepsilon'_{\mathtt{m}} = 1$
assert($\varepsilon'_{\mathtt{m}} < d_{min} \wedge \varepsilon'_{\mathtt{a}} < d_{min}$);
```

The variables `a` and `m` are replaced by their respective error weights and the comments on the right side of the code show one possible execution in which a bitflip is introduced in `a` and the bitflip propagates to `m`. The final checks are replaced by checks whether both errors are not masked, i.e., smaller than $d_{min}$. It it easy to verify (by hand or mechanically) that the assertion always holds.

To make things a little more interesting, let us extend the program by repeating the first statement:

```
m := m + a;
m := m + a;
check(m); check(a);
```

This program can no longer be verified using Abstraction Level 3, because a single bitflip in `a` at the beginning can result in $\varepsilon_{\mathtt{m}} = 2$ at the end, which is equal to $d_{min}$. However, we can use Abstraction Level 2 to show that all errors will be detected:

```
assume($\varepsilon_{\mathtt{a}} = 0 \wedge \varepsilon_{\mathtt{m}} = 0$);
$\varepsilon'_{\mathtt{a}}$ := flip$^a$($\varepsilon_{\mathtt{a}}$);               // $\varepsilon'_{\mathtt{a}} = 2,\ \varepsilon_{\mathtt{m}} = 0$
$\varepsilon'_{\mathtt{m}}$ := $\varepsilon_{\mathtt{m}}$ + $\varepsilon'_{\mathtt{a}}$;                // $\varepsilon'_{\mathtt{a}} = 2,\ \varepsilon'_{\mathtt{m}} = 2$
$\varepsilon''_{\mathtt{m}}$ := $\varepsilon'_{\mathtt{m}}$ + $\varepsilon'_{\mathtt{a}}$;                // $\varepsilon'_{\mathtt{a}} = 2,\ \varepsilon''_{\mathtt{m}} = 4$
assert($\mathrm{weight}(\varepsilon''_{\mathtt{m}}) < d_{min} \wedge \mathrm{weight}(\varepsilon'_{\mathtt{a}}) < d_{min}$);
```

The variables $\varepsilon_{\mathtt{a}}$ and $\varepsilon_{\mathtt{b}}$ now keep track of the precise faults. The comments show possible values for one execution with a bitflip on the second bit, which in the third line leads to a value with the third bit flipped. In general, injecting one bit flip in a variable and adding it to itself always results in a value with only one flipped bit, and such errors can be detected by a code with $d_{min} = 2$.

Extending our example once more, we get

```
m := m + a;
m := m + a;
m := m + a;
check(m); check(a);
```

An attempt to verify this program using Abstraction Level 2 fails, because the error weight of `m` can reach $d_{min}$ at the end. However, we can use Abstraction Level 1 to show that the check on `m` is still sufficient to find all faults in this program.

```
assume($\varepsilon_{\mathtt{a}} = 0 \wedge \varepsilon_{\mathtt{m}} = 0$);
$\varepsilon'_{\mathtt{a}}$ := flip$^a$($\varepsilon_{\mathtt{a}}$);               // $\varepsilon'_{\mathtt{a}} = 2,\ \varepsilon_{\mathtt{m}} = 0$
$\varepsilon'_{\mathtt{m}}$ := $\varepsilon_{\mathtt{m}}$ + $\varepsilon'_{\mathtt{a}}$;                // $\varepsilon'_{\mathtt{a}} = 2,\ \varepsilon'_{\mathtt{m}} = 2$
$\varepsilon''_{\mathtt{m}}$ := $\varepsilon'_{\mathtt{m}}$ + $\varepsilon'_{\mathtt{a}}$;                // $\varepsilon'_{\mathtt{a}} = 2,\ \varepsilon''_{\mathtt{m}} = 4$
$\varepsilon'''_{\mathtt{m}}$ := $\varepsilon''_{\mathtt{m}}$ + $\varepsilon'_{\mathtt{a}}$;               // $\varepsilon'_{\mathtt{a}} = 2,\ \varepsilon'''_{\mathtt{m}} = 6$
```

`assert(`$(\varepsilon'''_{\mathtt{m}} = 0 \vee \varepsilon'''_{\mathtt{m}} \mod A \neq 0) \wedge (\varepsilon'_{\mathtt{a}} = 0 \vee \varepsilon'_{\mathtt{a}} \mod A \neq 0)$`);`

Instead of checking only the weight of the error variables we check if the error variable is zero or a valid code word in the AN code. This is done by testing if the value is divisible by $A$. The comments again show one possible execution of the program. It is also easy to see that this is correct in general. The value of `m` at the end is `m`+3∗`a`; any error introduced at the beginning is also multiplied by 3. Error masking cannot occur, since 3 is not a factor of $A = 7$.

Abstraction Level 0 keeps the precise values of all variables. Essentially, this amounts to not using any abstraction. We will not go into details for this abstraction level, but of course, it is easy to come up with an example in which the concrete values of the variables are needed to show the program is secure.

## 4 Invariants

The abstract program defined in the last section can be passed to a model checker as is. However, such programs may be difficult for off-the-shelf model checkers to handle, especially in the presence of loops. It may, however, be easy to generate loop invariants for the classes of faults that we use, thus reducing the verification of an annotated program to the verification of a set of loop-free code segments in order to reduce the number of runtime checks.

Let us assume our annotated program $P_f$ contains a loop body $L$ that uses a set of variables $V = \{v_1, \ldots, v_n\}$ with the associated error variables $\varepsilon_V = \{\varepsilon_1, \ldots, \varepsilon_n\}$ and let $E \subseteq \mathbb{N}^n$ be the set of possible values for $\varepsilon_V$. Without loss of generality, let us assume that at the end of each loop iteration, we check variables $\{v_1, \ldots, v_k\}$ for a detectable error.

We will assume that we have an error specified by $\varphi_\varepsilon$ on the level of the loop that limits the bit flips to $l$, i.e., $l$ is the total number of bit flips that can be inserted during the execution of the loop. We denote the resulting value of variable $v_i$ (error $\varepsilon_i$) of executing $L$ with variable values $a = (a_1, \ldots, a_n)$ and error values $e = (e_1, \ldots, e_n)$ and the total number of bitflips introduced in this iteration $b$ by $a'_i(a, e, b)$ ($e'_i(a, e, b)$, resp.).

**Definition 2.** *For a loop $L$, a set $E^* \subseteq E \times \mathbb{N}$ is an* error invariant *if the following two conditions hold.*

1. *For any $(e_1, \ldots, e_n, b) \in E^*$, we have $\bigwedge_{i \leq k} \mathrm{corr}(e_i)$ and $\bigwedge_{i > k} \neg\, \mathrm{masked}(e_i)$ and $b \leq l$.*
2. *For any $(e, b) \in E^*$, any valuation $a$ of $V$, and any number of new bitflips $b' \leq l - b$, one of two things holds*
   *(a) $\bigvee_{i \leq k} \mathrm{detect}(e'_i(a, e, b'))$ or*
   *(b) $(e'_1(a, e, b'), \ldots, e'_n(a, e, b'), b + b') \in E^*$.*

Thus, if we start the loop body with the errors in the checked variables equal to zero and no masked errors, we know that at the end of the loop the program either terminates because it finds an error, or there are no masked errors. In addition to the errors on the variables the invariant also tracks the number of introduced bitflips and limits them to conform to the program wide fault model.

We will consider a loop to be correct if it has an error invariant, noting that if required, we can check for detectable error on variables $v_{k+1}, \ldots, v_n$ after the loop has finished.

**Theorem 2.** *Assume $E^*$ is an error invariant for a loop with body $L$. If $L$ is executed with error values and introduced bitfilps in $E^*$ and at the end either an error is raised or the values are in $E^*$ then executing the loop* `while(*): { L }` *with values in $E^*$ either results in raising an error or after the loop all values are in $E^*$.*

The general definition of invariants is independent of the abstraction level, but the invariants differ in the actual value of $E^*$. The main challenge is to find a good $E^*$. Many programs can be verified by using a few simple invariants.

– If all variables are checked at every loop iteration, we use $\{(0, \ldots, 0, b) \mid b \in [0, l]\}$ as a (candidate) error invariant.
– For Abstraction Level 3 we can use the invariant that all unchecked error variables are below $d_{min}$. The same can be done for Abstraction Level 2, but in this case we require that the Hamming weight of all variables be below $d_{min}$.
– For Abstraction Level 1 we can define $E^*$ as the set of detectable errors, according to the used code.
– Another stricter version for Abstraction Level 1 is to restrict the values to only what can be introduced with a single error injection and no accumulation.

These invariants assume a fault model that limits the amount of fault injections for one program execution. The invariants can be adapted to support other fault models. For instance only the number of bitflips per loop iteration could be bounded, without an upper limit for the whole execution.

*Example 5.* We use a variant of the example from the previous section to demonstrate our invariants. A simple multiplication algorithm can be build from repeated addition. The following code multiples `a` and `b` and stores the result in `m`.

```
m := 0;
while(i<b):
    i := i + 1;
    m := m + a;
    check(m); check(a);
```

The variables `i` and `b` are assumed to be checked by the control flow protection. We can therefore obtain the following program with abstracted control flow.

```
m := 0;
while(*):
    m := m + a;
    check(m); check(a);
```

Here both variables `m` and `a` are checked at the end of the loop body. Using Abstraction Level 2, we keep track of the errors and check their weights, resulting in the following program. In this case we can use the invariant that both $\varepsilon_{\mathtt{a}}$ and $\varepsilon_{\mathtt{m}}$ are zero, which gives the following program $P_{fa}$ using Definition 2.

```
assume($\varepsilon_{\mathtt{a}} = 0 \wedge \varepsilon_{\mathtt{m}} = 0$);
$\varepsilon'_{\mathtt{a}}$ := flip$^a$($\varepsilon_{\mathtt{a}}$);
$\varepsilon'_{\mathtt{m}}$ := $\varepsilon_{\mathtt{m}}$ + $\varepsilon_{\mathtt{a}}$;
assert($0 < \text{weight}(\varepsilon'_{\mathtt{m}}) < d_{min} \vee 0 < \text{weight}(\varepsilon'_{\mathtt{a}}) < d_{min} \vee \varepsilon'_{\mathtt{m}} = 0 \wedge \varepsilon'_{\mathtt{a}} = 0$);
```

We assume that both errors are zero at the start and we check that we can find potential errors in the variables after executing the loop body.

Suppose we want to only check one of the variables. Using Abstraction Level 1 and checking only `m`, we can define an invariant $\text{inv}(\varepsilon_{\mathtt{a}})$ that does not allow masked values. We obtain the program:

```
assume($\varepsilon_{\mathtt{m}} = 0 \wedge \text{inv}(\varepsilon_{\mathtt{a}})$);
$\varepsilon'_{\mathtt{a}}$ := flip$^a$($\varepsilon_{\mathtt{a}}$);
$\varepsilon'_{\mathtt{m}}$ := $\varepsilon_{\mathtt{m}}$ + $\varepsilon'_{\mathtt{a}}$;
assert($\text{detect}(\varepsilon'_m) \vee \varepsilon'_m = 0 \wedge \text{inv}(\varepsilon_{\mathtt{a}})$)
```

The last line in the listing can be realized by checking $\varepsilon'_{\mathtt{m}} \neq 0 \wedge \varepsilon'_{\mathtt{m}} \mod A \neq 0 \vee \varepsilon'_{\mathtt{m}} = 0 \wedge \text{inv}(\varepsilon'_{\mathtt{a}})$. For the invariant on $\varepsilon_{\mathtt{a}}$ we could use $\varepsilon_{\mathtt{a}} = 0$. This invariant holds, because any bitflip that is introduced in the second line will be found by the check on `m`.

## 5 Experimental Results

In order to evaluate how the overhead of runtime checks can be minimized using our method, we ran our technique on two examples, The CORDIC algorithm for numerical trigonometry [Vol59] and a Fibonacci number generator [Bon02]. These algorithms contain a loop in which almost all of the work is done, so that small performance improvements can have a large impact on the overall performance of the programs. To further reduce the number of required runtime check we also consider variants of these programs where the loop has been unrolled $n$ times. In these cases, checks are only inserted every $n$ iterations. We also use these two algorithms to compare the static verification results and performance of our approaches.

In our experiments, we used CPAchecker version 1.9 [BK11] running on a laptop with an Intel i5-6200U CPU under Ubuntu 18.04 with 12 GB of RAM. To verify invariants, we use the following settings in CPAchecker: MATHSAT5 solver, disabled outputs and disabled Java assertions. For comparison, we also verify the abstract programs when leaving the loops intact and not introducing invariants. Here we use the same settings with CPAchecker's k-induction configuration. [1]

[1] Our scripts are available at `https://extgit.iaik.tugraz.at/scos/rv20-fault-injection-checks`.

**Listing 1.** CORDIC program and abstraction

```
// concrete program
for (k in 0 to n):
  if (theta >=0 ):
    t_cosin := cosin - (sin>>k);
    t_sin := sin + (cosin>>k);
    theta := theta - table[k];
  else:
    t_cosin := cosin + (sin>>k);
    t_sin := sin - (cosin>>k);
    theta := theta + table[k];
  cosin := t_cosin;
  sin := t_sin;


// abstraction
while(*):
  if (*):
    (cosin, sin) := (cosin - (sin>>k)), (sin + (cosin>>k));
  else:
    (cosin, sin) := (cosin + (sin>>k)), (sin - (cosin>>k));
```

## 5.1 CORDIC

The CORDIC algorithm is used to calculate sine and cosine on devices without hardware support. It only requires addition, subtraction and right shifts. The results of the algorithm become more precise the more iterations are performed. Listing 1 shows an implementation for fixpoint arithmetic in two versions, first the original program and second the program with abstract control flow. The variable `table` refers to a precomputed array of constants that has been omitted from this listing. The abstract version of the program no longer contains the variables `n` and `theta` as they are already included in the control flow protection. The variable `k` is also checked as part of the control flow and we can assume that it does not contain errors. This makes the shifts conform to the error homomorphism property.

Our error assumption is that at most one arithmetic error is injected during the execution of the program. Our baseline comparison is to check each variable after each loop iteration. Using our abstractions we can show that it is sufficient to only check the variables every three loop iterations without reducing the fault resilience of the program, which reduces the runtime overhead by factor three.

Table 3 shows the experimental results for the CORDIC algorithm. It shows the abstraction level, the number of iterations of the loop that are performed before the variables are checked for errors, whether the technique can prove the approach correct or not and how much time it needs. The latter two categories are presented both for the techniques using invariants and for a plain run of CPAchecker. The program is protected by an AN-code where $A$ is the prime number 7919, which results in $d_{min} = 4$. Using Abstraction Level 3 (only tracking the weight of the error) we can prove that performing checks every second

**Table 3.** CORDIC verification results

| | | invariants | | loops | |
|---|---|---|---|---|---|
| abstr. lvl | iterations | success | time [s] | success | time [s] |
| 3 | 1 | ✓ | 2.71 | ✓ | 3.43 |
| | 2 | ✓ | 3.19 | ✓ | 4.78 |
| | 3 | ✗ | 3.77 | ✗ | 6.82 |
| 2 | 1 | ✗ | 4.37 | ✗ | 4.47 |
| 1 | 1 | ✓ | 9.25 | ✓ | 15.26 |
| | 2 | ✓ | 224.00 | ✓ | 200.33 |
| | 3 | ✓ | 2649.86 | ? | >3600 |
| | 4 | ? | >3600 | ? | >3600 |

**Listing 2.** Fibonacci program

```
(a, b) := (1, 1);
while(*):
    (a, b) := (a+b, a);
    check(a); check(b);
```

iteration is sufficient. However, this abstraction level is not precise enough to verify that we can perform three iterations of the loop before checking the variables. All these checks are completed in under four seconds.

A more precise abstraction allows us to prove programs with fewer checks, at the cost of a longer verification time. We note that Abstraction Level 2 is unsuitable for this specific program. Testing the Hamming weight instead of the arithmetic weight performs worse than using Abstraction Level 3. However, calculating the arithmetic weight during model checking is too expensive.

With Abstraction Level 1 we are able to establish that checks every three loop iterations are sufficient. This takes around 45 minutes, significantly longer than using the simpler abstraction. The runtime overhead of the checks, however, is reduced by a further 33%. Although the runtime differences between a plain run of CPAchecker and a run using invariants are not large, the most efficient configuration (two checks for every three iterations) can only be proved using invariants.

### 5.2 Fibonacci

As a second case study we analyze a Fibonacci number generator. The program consists of a loop and two variables `a` and `b`. We compare our techniques based on the static verification time and the number of required runtime checks. To do this, we vary both the number of iterations before checks are performed and the variables that are checked. The program is protected by an AN code with $A = 13$ and $d_{min} = 2$. The error assumption is that at most one arithmetic error is injected during the execution of the program. Our baseline comparison is a check on variables `a` and `b` after every iteration of the loop, giving us two checks per iteration. The code of the Fibonacci program with abstracted control flow is shown in Listing 2.

**Table 4.** Fibonacci experimental results

| | configuration | | | invariants | | loops | |
|---|---|---|---|---|---|---|---|
| abstr lvl | checked vars | iter | checks/iter | success | time [s] | success | time [s] |
| 3 | a,b | 1 | 2 | ✓ | 2.78 | ✓ | 3.03 |
| | a,b | 2 | 1 | ✗ | 2.81 | ✗ | 3.54 |
| | a | 1 | 1 | ✓ | 2.45 | ✓ | 3.81 |
| | a | 2 | 0.5 | ✗ | 2.75 | ✗ | 3.51 |
| 2 | a,b | 1 | 2 | ✓ | 3.45 | ✓ | 4.48 |
| | a,b | 2 | 1 | ✓ | 6.65 | ✓ | 10.16 |
| | a,b | 3 | 0.67 | ✗ | 5.54 | ✗ | 9.34 |
| | a | 1 | 1 | ✓ | 3.60 | ✓ | 6.33 |
| | a | 2 | 0.5 | ✓ | 7.80 | ✓ | 17.61 |
| | a | 3 | 0.33 | ✗ | 7.01 | ✗ | 9.41 |
| 1 | a,b | 1 | 2 | ✓ | 4.32 | ✓ | 8.76 |
| | a,b | 2 | 1 | ✓ | 6.57 | ✓ | 16.00 |
| | a,b | 3 | 0.67 | ✓ | 15.99 | ✓ | 46.62 |
| | a,b | 4 | 0.5 | ✓ | 43.92 | ✓ | 56.80 |
| | a,b | 5 | 0.4 | ✓ | 34.08 | ✓ | 190.94 |
| | a,b | 6 | 0.33 | ✗ | 38.85 | ✗ | 82.52 |
| | a | 1 | 1 | ✓ | 4.72 | ✓ | 15.26 |
| | a | 2 | 0.5 | ✓ | 11.36 | ✓ | 88.84 |
| | a | 3 | 0.33 | ✓ | 21.14 | ? | >600 |
| | a | 4 | 0.25 | ✓ | 132.25 | ? | >600 |
| | a | 5 | 0.2 | ✓ | 121.51 | ? | >600 |
| | a | 6 | 0.17 | ✗ | 14.06 | ✗ | 85.99 |

Table 4 shows the results of the Abstraction Levels 3 to 1. As before, we used both the approach with invariants and a vanilla run of CPAchecker. When checking only one variable we use the error invariant that the unchecked variable has a error weight less than $d_{min}$ for both Abstraction Levels 2 and 3. The number of checks per iteration is our final measure of runtime overhead.

We can observe that lower levels of abstraction allow us to verify programs with fewer runtime checks. When using Abstraction Level 3 we need at least one check per loop iteration on average. The verification time is around two to four seconds and using invariants performs slightly better than loops.

Moving to Abstraction Level 2 allows us to reduce the number of runtime checks per iteration to 0.5 checks when checking only one variable. The verification time increases, but is still relatively low.

Abstraction Level 1 provides the greatest benefits in terms of reducing the runtime overhead of the program. It allows us to reduce the required checks to only one check in every 5 iterations, an improvement of a factor of 10 over the original. These cases could not be verified using plain CPA within ten minutes.

For this algorithm, the final reduction in runtime overhead for checks is a factor of 10.

**Table 5.** Fibonacci encoding parameter selection

| A | checks/iter | checked vars | max iter | time [s] |
|---|---|---|---|---|
| 7 | 0.33 | a,b | 6 | 61.03 |
| | 0.2 | a | 5 | 93.39 |
| | 0.25 | b | 4 | 36.43 |
| 10 | 0.67 | a,b | 3 | 17.05 |
| | 0.33 | a | 3 | 12.36 |
| | 0.33 | b | 3 | 22.53 |
| 11 | 0.25 | a,b | 8 | 68.07 |
| | 0.2 | a | 5 | 74.20 |
| | 0.25 | b | 4 | 39.45 |
| 13 | 0.4 | a,b | 5 | 33.02 |
| | 0.2 | a | 5 | 127.20 |
| | 0.25 | b | 4 | 27.34 |
| 17 | 0.29 | a,b | 7 | 189.06 |
| | 0.2 | a | 5 | 472.71 |
| | 0.25 | b | 4 | 29.55 |

**Finding an Optimal Value for *A*.** As a second experiment on the Fibonacci program, we used invariants and Abstraction Level 1 to search for a good encoding parameter $A$ and the optimal placement of runtime checks. We tried five different values for $A$: 7, 10, 11, 13, and 17 that all have the same $d_{min}$ of 2. Three patterns for placing checks are explored: checking both the variables `a` and `b`, only checking `a` and only checking `b`. In all cases we maximize the number of loop iterations by increasing the iterations until the verification fails for the first time.

The results of this experiment are presented in Table 5. The maximum number of loop iterations between checks varies greatly based on the used encoding parameter. For $A = 10$ the program can only perform three iterations before it needs to check the variables, whereas for $A = 11$ we can do eight iterations if both variables are checked. The smallest runtime overhead can be achieved by using one of the prime numbers and performing a check on the variable `a` every five loop iterations. This results in only 0.2 checks per iteration, a significant improvement over the 2 checks per iteration from the naive check placement. As multiple coding parameters can achieve the same low runtime overhead we can look at the memory overhead as a tiebreaker. A smaller encoding parameter also results in a smaller memory overhead in the protected program. Thus, the most runtime efficient protection for this program is to use $A = 7$ and place a check on `a` every fifth iteration.

## 6 Conclusions

We have presented a method to analyze the necessity of runtime checks in programs using error correcting codes to achieve resilience against fault injections. Our method uses a combination of novel abstractions and simple recipes for loop invariants to achieve scalable verification times. We have shown that for simple examples we can reduce the overhead of runtime checks by factor of up to 10.

In future work, we will look at the use of different error detection codes, and we will consider combinations of secure hardware and software design to shield against fault injections.

**Acknowledgements.** We would like to thank Stefan Mangard and Robert Schilling for contributing their expertise on error correcting codes. We gratefully acknowledge the support of Graz University of Technology through the LEAD Project "Dependable Internet of Things in Adverse Environments".

## References

BBKN12. A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

BDL97. Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *EUROCRYPT '97*, pages 37–51, 1997.

BK11. Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *Lecture Notes in Computer Science*, volume 6806 LNCS, pages 184–190, 2011.

Bon02. Leonardo Bonacci. *Liber Abaci*. 1202.

Dia55. Joseph M Diamond. Checking Codes for Digital Computers. *Proceedings of the IRE*, 43(4):483–490, 1955.

Gar66. Harvey L Garner. Error Codes for Arithmetic Operations. *IEEE Trans. Electronic Computers*, 15(5):763–770, 1966.

Ham50. R. W. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 29(2):147–160, 4 1950.

KSBM19. Anja F. Karl, Robert Schilling, Roderick Bloem, and Stefan Mangard. Small faults grow up - Verification of error masking robustness in arithmetically encoded programs. In *Verification, Model Checking, and Abstract Interpretation 2019*, pages 183–204. Springer, 2019.

LRT12. Victor Lomné, Thomas Roche, and Adrian Thillard. On the need of randomness in fault attack countermeasures - application to AES. In Guido Bertoni and Benedikt Gierlichs, editors, *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, September 9, 2012*, pages 85–94. IEEE Computer Society, 2012.

MAN$^+$18. Lauren De Meyer, Victor Arribas, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. M&M: Masks and macs against physical attacks. Cryptology ePrint Archive, Report 2018/1195, 2018. `https://eprint.iacr.org/2018/1195`.

Mas64. James L Massey. Survey of residue coding for arithmetic errors. *International Computation Center Bulletin*, 3(4):3–17, 1964.

MER05. Shubhendu S Mukherjee, Joel Emer, and Steven K Reinhardt. The soft error problem: An architectural perspective. In *11th International Symposium on High-Performance Computer Architecture*, pages 243–247, 2005.

MS09. Marcel Medwed and Jrn Marc Schmidt. Coding schemes for arithmetic and logic operations - How robust are they? In *Lecture Notes in Computer Science*, volume 5932 LNCS, pages 51–65, 2009.

Pet58. W. W. Peterson. On Checking an Adder. *IBM Journal of Research and Development*, 2(2):166–168, 4 1958.

Rao70. Thammavarapu R N Rao. Biresidue Error-Correcting Codes for Computer Arithmetic. *IEEE Transactions on Computers*, 19(5):398–402, 1970.

RG71. Thammavarapu R N Rao and Oscar N Garcia. Cyclic and Multiresidue Codes for Arithmetic Operations. *IEEE Trans. Information Theory*, 17(1):85–91, 1971.

SFES18. Okan Seker, Abraham Fernandez-Rubio, Thomas Eisenbarth, and Rainer Steinwandt. Extending glitch-free multiparty protocols to resist fault injection attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):394–430, 2018.

SWM18. Robert Schilling, Mario Werner, and Stefan Mangard. Securing conditional branches in the presence of fault attacks. In *2018 Design, Automation & Test in Europe Conference & Exhibition*, pages 1586–1591. IEEE, 2018.

Vol59. Jack Volder. The cordic computing technique. In *Papers presented at the the March 3-5, 1959, Western Joint Computer Conference*, pages 257–261, 1959.

WUSM18. Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. Sponge-based control-flow protection for IoT devices. In *2018 IEEE European Symposium on Security and Privacy*, pages 214–226, 2018.

YSW18. Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. Fault attacks on secure embedded software: Threats, design, and evaluation. *J. Hardware and Systems Security*, 2(2):111–130, 2018.