Model-Based Diagnosis with Multiple Observations: A Unified Approach for C Software and Boolean Circuits

Pedro Orvalho[1a], Marta Kwiatkowska[a], Mikoláš Janota[b], Vasco Manquinho[c]

[a] *Department of Computer Science, University of Oxford, Oxford, United Kingdom*
[b] *Czech Technical University in Prague, Prague, Czech Republic*
[c] *INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal*

Abstract

Debugging is one of the most time-consuming and expensive tasks in software development and circuit design. Several formula-based fault localisation (FBFL) methods have been proposed, but they fail to guarantee a set of diagnoses across all failing tests or may produce redundant diagnoses that are not subset-minimal, particularly for programs/circuits with multiple faults.

This paper introduces CFaults, a novel fault localisation tool for C software and Boolean circuits with multiple faults. CFaults leverages Model-Based Diagnosis (MBD) with multiple observations and aggregates all failing test cases into a unified Maximum Satisfiability (MaxSAT) formula. Consequently, our method guarantees consistency across observations and simplifies the fault localisation procedure. Experimental results on three benchmark sets, two of C programs, TCAS and C-Pack-IPAs, and one of Boolean circuits, ISCAS85, show that CFaults is faster at localising faults in C software than other FBFL approaches such as BugAssist, SNIPER, and HSD. On the ISCAS85 benchmark, CFaults is generally slower than HSD; however, it localises faults in only 6% fewer circuits, demonstrating that it remains competitive in this domain. Furthermore, CFaults produces only subset-minimal diagnoses of faulty statements, whereas the other approaches tend to enumerate redundant diagnoses (e.g., BugAssist and SNIPER).

Keywords:
Model-Based Diagnosis, Fault Localisation, Formula-based Fault Localisation, Debugging, Maximum Satisfiability.

[1] PO is now affiliated with IIIA-CSIC, Spain. The work was conducted while he was at the University of Oxford.

1. Introduction

Localising system faults has long been recognised as one of the most time-consuming and expensive tasks in Artificial Intelligence (AI) [1, 2, 3, 4, 5]. Given a buggy system, *fault localisation (FL)* is the process of identifying the locations in the system that may cause faulty behaviour (bugs) [1]. More specifically, given a faulty system and a set of observations (i.e., failing test cases), *formula-based fault localisation (FBFL)* methods encode the localisation problem as a series of constraint optimisation problems to identify a minimal set of faulty statements (diagnoses) within the system. These FBFL approaches leverage the theory of *Model-Based Diagnosis (MBD)* [1], which has been applied to restore consistency across several domains, including Boolean circuits [2, 3, 6, 7, 8, 9], C software [10, 11, 5], knowledge bases [12, 13, 14], and spreadsheets [15, 16]. Typically, FBFL methods compute a minimal diagnosis by considering each failing test case individually rather than simultaneously across all failing test cases. Moreover, most of these methods [4, 10, 8, 11, 9, 5] usually enumerate all *Minimal Correction Subsets (MCSes)* [17] of a *Maximum Satisfiability* (MaxSAT) formula to cover all possible diagnoses, and then apply different aggregation mechanisms to determine the minimal diagnosis that explains the system inconsistency.

For instance, BugAssist [10, 4], a prominent FBFL tool for C software, implements a ranking mechanism for bug locations. As explained in Section 3.1, for each failing test, BugAssist enumerates all diagnoses of a MaxSAT formula corresponding to bug locations. Subsequently, BugAssist ranks diagnoses based on their frequency of appearance in each failing test. Other FBFL tools, like SNIPER [11], also enumerate all diagnoses for each failing test. However, as explained in Section 3.2, the set of SNIPER's diagnoses is obtained by taking the Cartesian product of the diagnoses gathered using each failing test. As a result, while FBFL methods can determine minimal diagnoses per failing test, BugAssist cannot guarantee a minimal diagnosis considering all failing tests, and SNIPER may enumerate a significant number of redundant diagnoses that are not minimal [9]. These limitations may pose challenges for C programs with multiple faulty statements, as shown in Example 1.

Example 1 (Motivation). *Consider the C program presented in Listing 1, which aims to determine the maximum among three given numbers. However,*

Listing 1: Faulty program example. Faulty lines: {5,8,11}.

```
1  int main(){
2    // finds maximum of 3 numbers
3    int f,s,t;
4    scanf("%d%d%d",&f,&s,&t);
5    if (f < s && f >= t)
6      // fix: f >= s
7      printf("%d",f);
8    if (f > s && s <= t)
9      // fix: f < s and s >= t
10     printf("%d",s);
11   if (f > t && s > t)
12     // fix: f < t and s < t
13     printf("%d",t);
14
15   return 0;
16 }
```

	Input			Output
t_0	1	2	3	3
t_1	6	2	1	6
t_2	-1	3	1	3

Table 1: Test-suite.

	BugAssist	SNIPER
#Diagnoses t_0	8	8
#Diagnoses t_1	21	21
#Diagnoses t_2	9	9
#Total Unique Diagnoses	32	1297
Final Diagnosis	{4,13}	{5,8,11}

Table 2: Number of diagnoses (faulty statements) generated by BugAssist [4] and SNIPER [11] per test.

based on the test suite shown in Table 1, the program is faulty, as its output differs from the expected. The set of minimally faulty lines in this program is {5, 8, 11}, as all three `if`*-conditions are incorrect according to the test suite. Fixing any subset of these lines would be insufficient to repair the program. One possible fix is to replace all these conditions with the suggested fixes in lines {6, 9, 12}.*

In a typical FBFL approach for C software, the minimal set of statements identified as faulty might include, for example, lines 4 and 5. Removing the `scanf` *statement and an* `if`*-statement would allow an FBFL tool to assign any value to the input variables in order to always produce the expected output. However, considering an approach that prioritises identifying faulty statements within the program's logic before evaluating issues in the input/output statements (such as* `scanf` *and* `printf`*), one might identify lines {5, 8, 11} as the faulty statements. When applying BugAssist's and SNIPER's approach on the program in Listing 1 with the described optimisation criterion and utilising the inputs/outputs detailed in Table 1 as specification, distinct sets of faults are identified for each failing test. Table 2 presents the diagnosis (set of faulty lines) produced by each tool, along with the number of diagnoses enumerated for each failing test case and the total number of unique diagnoses after aggregating the diagnoses from all tests, using each tool's respective method.*

In the case of BugAssist, diagnoses are prioritised based on their occurrence frequency. Consequently, BugAssist yields 32 unique diagnoses and selects {4, 13} since this diagnosis is identified in every failing test. In

contrast, SNIPER computes the Cartesian product of all diagnoses, resulting in 1297 unique diagnoses. Note that BugAssist*'s diagnoses may not adequately identify all faulty program statements. Conversely, SNIPER's final diagnosis {5, 8, 11} is minimal, even though it enumerates an additional 1296 diagnoses. Hence, existing FBFL methods do not ensure a minimal diagnosis across all failing tests (e.g.,* BugAssist*) or may produce an overwhelming number of redundant sets of diagnoses (e.g., SNIPER), especially for programs with multiple faults.*

This paper tackles this challenge by formulating the fault localisation (FL) problem as a single optimisation problem, as described in Section 4. We leverage MaxSAT and the theory of *Model-Based Diagnosis (MBD)* [1, 9, 18], integrating all failing test cases simultaneously. This approach enables the generation of only minimal diagnoses that identify all faulty components within a system, in our case either a C program or a Boolean circuit. Furthermore, we implement the MBD problem with multiple test cases in CFaults, a fault localisation tool designed to *see faults in C* programs and Boolean *C*ircuits, presented in Section 5.

For C programs, CFaults first unrolls and instruments the code at the program level, ensuring independence from the bounded model checker. It then calls CBMC [19], a well-known bounded model checker for C, to generate a trace formula. Finally, CFaults encodes the problem into MaxSAT to identify the minimal set of diagnoses corresponding to the buggy statements.

For Boolean circuits, CFaults first unrolls and instruments the given circuit for each failing observation. The resulting unrolled and instrumented circuit is then directly translated into a unified MaxSAT formula. A MaxSAT solver is invoked to compute the minimal diagnosis that restores consistency between the Boolean circuit and all observations.

In addition, we adapt the *hitting set dualisation* (HSD) [9] algorithm, the current publicly available state-of-the-art FBFL method for Boolean circuits, to perform fault localisation in C software.

Experimental results presented in Section 6.1, on two benchmarks of C programs, TCAS [20] (industrial) and C-Pack-IPAs [21] (programming exercises), show that CFaults consistently detects only minimal sets of diagnoses and outperforms other FBFL methods such as BugAssist, SNIPER, and HSD in terms of speed. In contrast, SNIPER and BugAssist either generate an overwhelming number of redundant diagnoses or fail to produce the minimal sets required to repair each program. Moreover, CFaults identi-

fies faults in 31% more programs than BugAssist, 36% more programs than SNIPER, and 15% more programs than HSD. Section 6.2 further presents experiments using CFaults and HSD to localise bugs in Boolean circuits on the ISCAS85 [22] benchmark, showing that while CFaults is generally slower than HSD, it localises only 6% fewer faulty circuits, thereby remaining competitive for Boolean circuit fault localisation. Finally, Section 8 presents several use cases where CFaults has been successfully applied, Section 9 reviews related work, and the paper concludes in Section 10.

To summarise, the contributions of this work are:

- We address the fault localisation problem in C programs and Boolean circuits using a Model-Based Diagnosis (MBD) approach that considers multiple failing test cases and formulates the problem as a unified optimisation task.
- Our MBD approach will be made publicly available in an upcoming release of CFaults [23][2], our fault localisation tool that unrolls and instruments C programs at the code level, thereby decoupling it from any specific bounded model checker. That release will also support fault localisation in Boolean circuits.
- Experiments using CFaults on two sets of C programs (TCAS and C-Pack-IPAs) show that CFaults is faster, and localises faults in more programs than the other localisation approaches.
- CFaults produces only subset-minimal diagnoses, unlike other state-of-the-art formula-based fault localisation (FBFL) tools, such as BugAssist and SNIPER.
- On ISCAS85, a benchmark of Boolean circuits, CFaults is generally slower than HSD. However, it localises faults in only 6% fewer circuits, demonstrating that it remains competitive in this domain.

Contributions Beyond the Conference Version. This article substantially extends the conference paper *"CFaults: Model-Based Diagnosis for Fault Localisation in C with Multiple Test Cases"* [5] by introducing new theoretical developments, improved techniques, and a broader experimental evaluation.

[2] https://github.com/pmorvalho/CFaults

First, the original method, designed exclusively for C software, is *generalised* to also localise faults in Boolean circuits, providing a single formula-based framework capable of handling both C programs and Boolean circuits under multiple failing observations. *Second*, several relaxation mechanisms for program statements are revised to correctly address specific issues, such as non-terminating loops, and uninitialised global variables, thus ensuring complete fault localisation under our model-based diagnosis (MBD) approach (see Section 5). *Third*, we include a detailed empirical comparison with HSD [9], the publicly available state-of-the-art formula-based fault-localisation method for Boolean circuits, highlighting the strengths and limitations of our unified approach. *Fourth*, the background and theoretical exposition are significantly expanded to offer a more comprehensive discussion of MBD, including additional explanations of core algorithms and related approaches, such as BugAssist, SNIPER, and HSD. *Finally*, the experimental evaluation is significantly extended: the assessment on C software now covers the full C-Pack-IPAs benchmark [21], increasing from roughly 500 to nearly 1500 higher-complexity programs; additionally, we report new experiments on Boolean circuits using the ISCAS85 benchmark [22].

2. Preliminaries

This section provides definitions that are used throughout the paper.

Definition 1 (Boolean Satisfiability (SAT)). *The Boolean Satisfiability (SAT) problem is the decision problem for propositional logic [24].*

A propositional formula in Conjunctive Normal Form (CNF) is a conjunction of clauses where each clause is a disjunction of literals. A literal is a propositional variable x_i or its negation $\neg x_i$. Given a CNF formula ϕ, the SAT problem corresponds to deciding if there is an assignment to the variables in ϕ such that ϕ is satisfied or prove that no such assignment exists. When applicable, set notation will be used for formulas and clauses. A formula can be represented as a set of clauses (meaning its conjunction) and a clause as a set of literals (meaning its disjunction).

Example 2 (Satisfiable Formula). *Let ϕ denote a CNF formula with the following clauses: $\{(x_1), (\neg x_1 \vee x_2), (x_2 \vee x_3), (\neg x_1 \vee \neg x_3)\}$. In this case, the assignment $\{(x_1, 1), (x_2, 1), (x_3, 0)\}$ satisfies ϕ.*

Example 3 (Unsatisfiable Formula). *Let ϕ denote a CNF formula as follows: $\{(x_1), (\neg x_1 \vee x_2), (x_2 \vee x_3), (\neg x_1 \vee \neg x_3), (x_3)\}$. Since there is no assignment to the variables such that ϕ is satisfied, we say that ϕ is unsatisfiable.*

Definition 2 (SAT Solver Call). *Given a CNF formula ϕ, the result of the SAT solver call SAT(ϕ) is a triple (st, ν, ϕ_C), where st denotes the solver status (`SAT` or `UNSAT`). If the call is `SAT`, ν is a model of ϕ. Otherwise, $\phi_C \subseteq \phi$ contains an `unsatisfiable core`, that is, a subformula for which no assignment exists that satisfies all its clauses simultaneously [25].*

Example 4 (Unsatisfiable Core). *Let ϕ denote a CNF formula as follows: $\{(x_1), (\neg x_1 \vee x_2), (x_2 \vee x_3), (\neg x_1 \vee \neg x_3), (x_3)\}$. ϕ is unsatisfiable, and $\phi_C = \{(x_1), (\neg x_1 \vee \neg x_3), (x_3)\}$ is an unsatisfiable core since there is no assignment that satisfies all the clauses in ϕ_C simultaneously.*

Definition 3 (Maximum Satisfiability (MaxSAT)). *The Maximum Satisfiability (MaxSAT) problem is an optimisation version of the SAT problem. Given a CNF formula ϕ, the goal is to find an assignment that maximises the number of satisfied clauses in ϕ [26, 27].*

In partial MaxSAT, ϕ is split into hard clauses (ϕ_h) and soft clauses (ϕ_s). Given a formula $\phi = (\phi_h, \phi_s)$, the goal is to find an assignment that satisfies all hard clauses in ϕ_h while minimising the number of unsatisfied soft clauses in ϕ_s. Moreover, in the weighted version of the partial MaxSAT problem, each soft clause is assigned a weight, and the goal is to find an assignment that satisfies all hard clauses and minimises the sum of the weights of the unsatisfied soft clauses.

Example 5 (MaxSAT). *Consider the partial MaxSAT formula $\phi = (\phi_h, \phi_s)$ where $\phi_h = \{(x_1 \vee x_2), (\neg x_2 \vee x_3)\}$ and $\phi_s = \{(\neg x_1), (\neg x_3)\}$. In this case, the assignment $\{(x_1, 1), (x_2, 0), (x_3, 0)\}$ is an optimal assignment to ϕ since it only unsatisfies one clause in ϕ_s while satisfying all clauses in ϕ_h.*

Several MaxSAT algorithms rely on iterative calls to a SAT solver. In particular, *Core-guided MaxSAT algorithms* [25, 28, 29, 30] have proven highly effective for instances arising from real-world applications [28]. These algorithms exploit the ability of SAT solvers to identify *unsatisfiable cores*.

Definition 4 (Minimal Correction Subset (MCS)). *Given a MaxSAT formula $\phi = (\phi_h, \phi_s)$, a Minimal Correction Subset (MCS) κ of ϕ is a subset $\kappa \subseteq \phi_s$ where $\phi_h \cup (\phi_s \setminus \kappa)$ is satisfiable and, for all $c \in \kappa$, $\phi_h \cup (\phi_s \setminus \kappa) \cup \{c\}$ is unsatisfiable.*

Example 6 (MCS). *Consider the partial MaxSAT formula $\phi = (\phi_h, \phi_s)$ where $\phi_h = \{(x_1 \vee x_2), (x_2 \vee \neg x_3), (\neg x_2 \vee x_3)\}$ and $\phi_s = \{(\neg x_1), (\neg x_2), (\neg x_3)\}$. One possible MCS would be $C_1 = \{(\neg x_1)\}$. Note that $\phi_h \cup \phi_s$ is unsatisfiable, but $\phi_h \cup \phi_s \setminus C_1$ is satisfiable and C_1 is minimal. Another MCS of the formula is $C_2 = \{(\neg x_2), (\neg x_3)\}$.*

A dual concept of MCSes are *Minimal Unsatisfiable Subsets (MUSes)* [17, 31].

Definition 5 (Minimal Unsatisfiable Subset (MUS)). *Let $\phi = (\phi_h, \phi_s)$ denote a MaxSAT formula. A Minimal Unsatisfiable Subset (MUS) μ of ϕ is a subset $\mu \subseteq \phi_s$ where $\phi_h \cup \mu$ is unsatisfiable and, for all $c \in \mu$, $\phi_h \cup \mu \setminus \{c\}$ is satisfiable.*

Example 7 (MUS). *Consider the partial MaxSAT formula $\phi = (\phi_h, \phi_s)$ where $\phi_h = \{(x_1 \vee x_2), (x_2 \vee \neg x_3), (\neg x_2 \vee x_3)\}$ and $\phi_s = \{(\neg x_1), (\neg x_2), (\neg x_3)\}$. One possible MUS would be $\mu_1 = \{(\neg x_1), (\neg x_2)\}$. Note that $\phi_h \cup \mu_1$ is unsatisfiable and μ_1 is minimal. Another MUS is $\mu_2 = \{(\neg x_1), (\neg x_3)\}$.*

Definition 6 (Program). *A program is considered sequential, comprising standard statements such as assignments, conditionals, loops, and function calls, each adhering to their conventional semantics in C. A program is deemed to contain a bug when an assertion violation occurs during its execution with input I. Conversely, if no assertion violation occurs, the program is considered correct for input I. In cases where a bug is detected for input I, it is possible to define an error trace, representing the sequence of statements executed by program P on input I [19].*

Definition 7 (Trace Formula (TF)). *A Trace Formula (TF) is a propositional formula that is SAT iff there exists an execution of the program that terminates with a violation of an assert statement while satisfying all assume statements [10].*

For further information on TFs, interested readers are referred to [19, 32].

3. Model-Based Diagnosis With One Observation

The following definitions are commonly used in the *Model-Based Diagnosis (MBD)* theory [9, 1, 8]. A system description $\mathcal{P}$ is composed of a set of components $\mathcal{C} = \{c_1, \ldots, c_n\}$. Each component in $\mathcal{C}$ can be declared *healthy* or *unhealthy*. For each component $c \in \mathcal{C}$, $h(c) = 0$ if c is unhealthy, otherwise, $h(c) = 1$. As in prior works [9, 7], $\mathcal{P}$ is described by a CNF formula, where $\mathcal{F}_c$ denotes the encoding of component c:

$$\mathcal{P} \triangleq \bigwedge\nolimits_{c \in \mathcal{C}} (\neg h(c) \vee \mathcal{F}_c) \tag{1}$$

Observations represent deviations from the expected system behaviour. An observation, denoted as o, is a finite set of first-order sentences [1, 9], which is assumed to be encodable in CNF as a set of unit clauses. In this work, the failing test cases represent the set of observations.

A system $\mathcal{P}$ is considered faulty if there exists an inconsistency with a given observation o when all components are declared healthy. The problem of model-based diagnosis (MBD) aims to identify a set of components which, if declared unhealthy, restore consistency. This problem is represented by the 3-tuple $\langle \mathcal{P}, \mathcal{C}, o \rangle$, and can be encoded as a CNF formula:

$$\mathcal{P} \wedge o \wedge \bigwedge\nolimits_{c \in \mathcal{C}} h(c) \models \bot \tag{2}$$

For a given MBD problem $\langle \mathcal{P}, \mathcal{C}, o \rangle$, a set of system components $\Delta \subseteq \mathcal{C}$ is a diagnosis iff:

$$\mathcal{P} \wedge o \wedge \bigwedge\nolimits_{c \in \mathcal{C} \setminus \Delta} h(c) \wedge \bigwedge\nolimits_{c \in \Delta} \neg h(c) \not\models \bot \tag{3}$$

A diagnosis Δ is minimal iff no subset of Δ, $\Delta' \subsetneq \Delta$, is a diagnosis, and Δ is of minimal cardinality if there is no other diagnosis $\Delta'' \subseteq \mathcal{C}$ with $|\Delta''| < |\Delta|$.

A diagnosis is redundant if it is not subset-minimal [9].

To encode the Model-Based Diagnosis problem with one observation with partial MaxSAT, the set of clauses that encode $\mathcal{P}$ (1), and the set of clauses that encode observation o, represent the set of hard clauses. The soft clauses consist of unit clauses that aim to maximise the set of healthy components, i.e., $\bigwedge_{c \in \mathcal{C}} h(c)$ [33, 8]. This MaxSAT encoding of MBD enables enumerating minimum cardinality diagnoses and subset minimal diagnoses, considering a single observation. Furthermore, a minimal diagnosis is a minimal correction

Algorithm 1: BugAssist's Fault Localisation Algorithm [10].

Input: Program P, assertion p, failing tests $\{t_1, \ldots, t_k\}$
Output: Ranked list of unique MCSes based on frequency across failing tests

$\mathcal{M}_{all} \leftarrow []$ `// List of MCS sets for each test`
foreach *test* $t_i \in \{t_1, \ldots, t_k\}$ **do**
  $(\texttt{test}, \sigma) \leftarrow \mathsf{GenerateTrace}(P, t_i)$
  if $\sigma =$ *None* **then**
    continue `// Skip if no trace was found`
  $\Phi_H \leftarrow \texttt{test} \wedge p \wedge TF_1(\sigma)$
  $\Phi_S \leftarrow TF_2(\sigma)$
  $\mathcal{M}_i \leftarrow EnumerateMCSes(\Phi_H, \Phi_S)$ `// MCSes for this test`
  $\mathcal{M}_{all} \leftarrow \mathcal{M}_{all} \cup \{\mathcal{M}_i\}$
$\mathcal{R} \leftarrow \mathsf{VoteMCS}(\mathcal{M}_{all})$;
return $\mathcal{R}$

subset (MCS) of the MaxSAT formula. Given an inconsistent formula that encodes the MDB problem (2), a minimal diagnosis Δ satisfies (3), thereby making Δ an MCS of the MaxSAT formula. BugAssist [10], SNIPER [11], and other model-based diagnosis (MBD) tools for fault localisation in circuits [8, 33] encode with partial MaxSAT the localisation problem using a single failing observation.

3.1. BugAssist

BugAssist [4, 10], an FBFL approach for C software, prioritises diagnoses according to their frequency of occurrence. Algorithm 1 outlines BugAssist's localisation procedure. For each failing test case, BugAssist encodes the MBD problem, using a single observation (i.e., a failing test case), as a MaxSAT formula following the encoding in (2), and enumerates all minimal correction subsets (MCSes). It then invokes Algorithm 2 to rank each MCS based on its frequency across the failing test cases. The MCS that occurs most frequently and contains the fewest faulty components is ultimately returned to the user.

Algorithm 2: VoteMCS: Aggregate and Rank MCSes by Frequency and Size

Input: t sets of MCSes: $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_t$
Output: Sorted list $\mathcal{R}$ of unique MCSes

$\mathcal{F} \leftarrow \emptyset$ `// Map: MCS → frequency count`
foreach $\mathcal{M}_i$ *in* $\{\mathcal{M}_1, \ldots, \mathcal{M}_t\}$ **do**
  foreach *MCS* S *in* $\mathcal{M}_i$ **do**
    if $S \notin \mathcal{F}$ **then**
      $\mathcal{F}[S] \leftarrow 1$
    else
      $\mathcal{F}[S] \leftarrow \mathcal{F}[S] + 1$
$\mathcal{R} \leftarrow \mathsf{Keys}(\mathcal{F})$
Sort $\mathcal{R}$ by: (1) descending frequency $\mathcal{F}[S]$; and then (2) ascending set size $|S|$ to break ties;
return $\mathcal{R}$

Algorithm 3: SNIPER's SetCombine Function [11].

Input: A set of sets of MCSes $D = \{D_1, D_2, \ldots, D_n\}$
Output: Set of complete diagnoses $\mathcal{C}$

$\mathcal{C} \leftarrow \emptyset$
foreach *tuple* $(d_1, d_2, \ldots, d_n) \in D_1 \times D_2 \times \cdots \times D_n$ **do**
  $C \leftarrow \bigcup_{i=1}^{n} d_i$
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{C\}$
return $\mathcal{C}$

3.2. SNIPER

Similarly to BugAssist, SNIPER also enumerates all MCSes for each failing test case. The key difference between the two FBFL methods lies in how they aggregate this information to determine the most appropriate MCS to return to the user (line 10 in Algorithm 1). Unlike BugAssist, which treats each failing test case independently, SNIPER performs a Cartesian product over all generated MCSes, combining information across different failing test cases using Algorithm 3. This strategy allows SNIPER to compute a *subset-minimal aggregated diagnosis*, i.e., the smallest set of faulty

components that explains all failing test cases. However, achieving this diagnosis requires SNIPER to enumerate a large number of redundant diagnoses, which significantly increases computational overhead.

4. Model-Based Diagnosis with Multiple Observations

More recently, the MaxSAT encoding for MBD has been generalised to multiple inconsistent observations [9]. Let $\mathcal{O} = \{o_1, \ \ldots \ o_m\}$ be a set of observations. Each observation is associated with a replica $\mathcal{P}_i$ of the system $\mathcal{P}$. The system remains unchanged given different observations, where the components are replicated for each observation, but the healthy variables are shared. For a given observation o_i, a diagnosis is given by the following:

$$\mathcal{P}_i \wedge o_i \wedge \bigwedge\nolimits_{c \in \mathcal{C} \setminus \Delta} h(c) \wedge \bigwedge\nolimits_{c \in \Delta} \neg h(c) \not\models \bot \tag{4}$$

The goal is to find a minimal diagnosis $\Delta \subseteq \mathcal{C}$, such that Δ is a minimal set of components when deactivated the system becomes consistent with all observations $\mathcal{O} = \{o_1, \ \ldots \ o_m\}$. Moreover, when considering multiple observations, an aggregated diagnosis is a subset of components that includes one possible diagnosis for each given observation.

4.1. HSD

The HSD algorithm [9], presented in Algorithm 4, was proposed for fault localisation in systems based on multiple observations. It is based on *hitting set dualisation (HSD)* [6, 34, 35]. For each observation o_i, the algorithm computes minimal unsatisfiable subsets (MUSes) from the MaxSAT encoding defined in Equation (4). It then calculates a minimum hitting set Δ over the unsatisfiable cores (MUSes if core minimisation is applie) and checks whether Δ makes the system consistent with each observation individually. To compute all subset-minimal aggregated diagnoses for a faulty system $\mathcal{P}$, the algorithm performs at most m oracle calls for each minimum hitting set, where m is the number of observations. Each oracle call uses a distinct system replica based on Equation (4).

Note that, in line 9, if the minimum hitting set Δ of the MUSes is not an aggregated diagnosis, i.e., it does not constitute a diagnosis for at least one of the observations, the SAT oracle returns `UNSAT` together with an unsatisfiable core κ of the formula (see Section 2). The HSD algorithm can be run in two configurations: one that applies minimisation to the *unsatisfiable*

Algorithm 4: Hitting Set Dualisation Fault (HSD) Localisation Algorithm with Multiple Observations [9].

Input: System description $\mathtt{P}$, Observations $O = \{o_1, \ldots, o_m\}$
Output: All subset-minimal aggregated diagnoses $\mathcal{D} = \{\Delta_1, \Delta_2, \ldots\}$ and explanations $\mathcal{U} = \{U_1, U_2, \ldots\}$

1 $(P_1, \ldots, P_m) \leftarrow \mathsf{Encode}(\mathtt{P}, \mathtt{o}_1, \ldots, \mathtt{o}_m)$ `// m replicas of P and observations, encoded into m formulae (hard clauses)`
2 $\phi_S \leftarrow \{h(c) \mid c \in \mathcal{C}\}$ `// Healthy variables (soft clauses)`
3 $\mathcal{D} \leftarrow \emptyset$ `// Diagnoses`
4 **while** *true* **do**
5 $(\mathtt{status}, \Delta) \leftarrow \mathsf{MinHS}(\mathcal{U}, \mathcal{D})$ `// Minimal HS of diagnoses`
6 **if** *status* $=$ *false* **then**
7 **break**
8 **foreach** $i \in \{1, \ldots, m\}$ **do**
9 $(\mathtt{status}_i, \kappa) \leftarrow \mathsf{SAT}(P_i \cup (\phi_S \setminus \Delta))$
10 **if** *status_i* $=$ *false* **then**
11 $U \leftarrow \mathsf{Reduce}(\kappa)$ `// Extract MUS (explanation)`
12 $\mathcal{U} \leftarrow \mathcal{U} \cup \{U\}$
13 **break loop**
14 **if** *all* $P_i \cup (\phi_S \setminus \Delta)$ *are satisfiable* **then**
15 $\mathcal{D} \leftarrow \mathcal{D} \cup \{\Delta\}$
16 **foreach** $i \in \{1, \ldots, m\}$ **do**
17 **if** $\mathit{SAT}(P_i \cup \mathcal{D}) =$ *false* **then**
18 **return**

cores Δ in line 11, referred to as HSD-CM, and another that does not perform this minimisation, referred to simply as HSD. The core minimisation step consists of at least one additional oracle call that allows the algorithm to reduce the working formula $\mathcal{U}$. This helps to minimise memory usage, although a potential drawback is the increase in computation time caused by the extra(s) SAT call(s). Note that the number of cores computed by this algorithm corresponds directly to the number of iterations performed by the HSD algorithm.

Moreover, we would like to point out that HSD has only been evaluated on

Algorithm 5: Enumerate All MaxSAT Solutions

Input: System description P, Observations $O = \{o_1, \ldots, o_m\}$
Output: All subset-minimal aggregated diagnoses
$\mathcal{D} = \{\Delta_1, \Delta_2, \ldots\}$

$\mathcal{D} \leftarrow \emptyset$ // Diagnoses
optCost $\leftarrow$ None // Best known cost
$\phi_H \leftarrow \bigcup_{i=1}^{m} P_i \vee o_i$ // m replicas of P and observations, encoded into one formula (hard clauses)
$\phi_S \leftarrow \{h(c) \mid c \in \mathcal{C}\}$ // Soft clauses (healthy variables)
while *true* **do**
  $(\Delta, \texttt{cost}) \leftarrow \mathsf{SolveMaxSAT}(\phi_H, \phi_S)$
  if $\Delta =$ *None* **then**
    break // No more solutions
  if *optCost* $=$ *None* **then**
    optCost $\leftarrow$ cost // Initialise optimal cost
  if *cost* $>$ *optCost* **then**
    break // All MaxSAT solutions enumerated.
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{\Delta\}$
  Add blocking clause: $\bigvee_{h(c) \in \Delta} \neg h(c)$ to ϕ_H
return $\mathcal{D}$

Boolean circuits with single faults, and its publicly available implementation does not support FBFL for C programs.

4.2. Our approach

This paper encodes the fault localisation problem as a Model-Based Diagnosis with multiple observations using a single optimisation problem. We simultaneously integrate all failing test cases (observations) in a single MaxSAT formula. This approach allows us to generate only minimal diagnoses capable of identifying all faulty components within the system, in our case, a C program or a Boolean circuit. Algorithm 5 presents our approach.

Given m observations, $\mathcal{O} = \{o_1, \ldots, o_m\}$, a distinct replica of the system, denoted as $\mathcal{P}_i$, is required for each observation o_i. The hard clauses, ϕ_h, in our MaxSAT formulation correspond to each observation's encoding (o_i) and m system replicas, one for each observation, $\mathcal{P}_i$. Hence, $\phi_h = \bigwedge_{o_i \in \mathcal{O}} (\mathcal{P}_i \wedge o_i)$.

Additionally, we aim to maximise the set of healthy components. Therefore, the soft clauses are formulated as: $\phi_s = \bigwedge_{c \in \mathcal{C}} h(c)$. Thus, given the MaxSAT solution of (ϕ_h, ϕ_s), the set of unhealthy components ($h(c) = 0$), corresponds to a subset-minimal aggregated diagnosis, which also represents the smallest minimal diagnosis. This diagnosis is a subset-minimal of components that, when declared unhealthy (deactivated), make the system consistent with all observations, as follows:

$$\bigwedge\nolimits_{o_i \in \mathcal{O}} (\mathcal{P}_i \wedge o_i) \wedge \bigwedge\nolimits_{c \in \mathcal{C} \setminus \Delta} h(c) \wedge \bigwedge\nolimits_{c \in \Delta} \neg h(c) \not\models \bot \tag{5}$$

We assume that the system remains unchanged given different observations, where the components are replicated for each observation, but the healthy variables are shared. This is necessary because we analise all observations jointly, which can affect the component's behaviour.

4.3. Our Approach vs HSD

Our approach encodes the problem into a single MaxSAT formula, whereas HSD [9] divides it into m MaxSAT formulas, one for each observation. Moreover, for each minimum hitting set computed by HSD, m oracle calls are required to verify whether a diagnosis is consistent with all observations. In contrast, our method requires only a single MaxSAT call, which directly returns a cardinality minimal diagnosis that is, by definition, consistent with all observations, since they are all encoded within the same formula. Furthermore, the HSD algorithm has been evaluated solely on circuits with single faults under multiple observations and was not originally implemented for programs. We have adapted HSD to localise faults in C software, and in Section 6 we compare the performance of CFaults against HSD on both C programs and Boolean circuits.

5. CFaults: MBD with Multiple Observations for C Software and Boolean Circuits

This section presents CFaults, which is a model-based diagnosis (MBD) tool for fault localisation in C programs and Boolean circuits with multiple failing test cases. Unlike previous works, CFaults uses the approach proposed in Section 4.2, and C programs are relaxed at the code level, enabling users to leverage other bounded model checkers effectively. Figure 1 provides an overview of CFaults consisting of five steps: system unrolling,

system instrumentalisation, bounded model checking (CBMC), encoding to MaxSAT, and an Oracle (MaxSAT solver).

As explained in Section 5.1 and illustrated by the blue arrows in Figure 1, when CFAULTS receives a C program and a test suite as input, all processing steps are executed. In this case, CFAULTS formulates the MBD problem with multiple test cases as the 3-tuple $\langle \mathcal{P}_i, \mathcal{C}, \mathcal{O} \rangle$, where the observations $\mathcal{O}$ correspond to failing test cases (inputs and assertions), the components $\mathcal{C}$ denote the set of program statements, and the system description $\mathcal{P}_i$ is a trace formula derived from the unrolled and instrumented program. The program is instrumented at the code level using relaxation variables, referred to as *healthy variables*.

Furthermore, as detailed in Section 5.2 and indicated by the orange dashed arrows in Figure 1, when CFAULTS receives a Boolean circuit, `B`, and a test suite as input, only four steps are performed: unrolling the circuit, instrumenting the unrolled circuit, encoding it into MaxSAT, and calling an oracle. In this scenario, CFAULTS formulates the MBD problem as the 3-tuple $\langle B_i, \mathcal{C}, \mathcal{O} \rangle$, where the observations $\mathcal{O}$ consist of failing input–output test cases, the components $\mathcal{C}$ represent the set of gates in the Boolean circuit, and the system description B_i is a CNF formula encoding the unrolled and instrumentalised Boolean circuit `B`.

5.1. CFAULTS sees Faults in C Programs

Before unrolling the given C program, CFAULTS first runs the static analysers CPPCHECK [36] and CLANG-TIDY [37] to detect issues, such as, uninitialised variables and potential runtime errors (e.g., division by zero). If any issues are detected, CFAULTS immediately reports them to the user; otherwise, it proceeds to unroll the buggy program. We integrated static analysers after our user study [38] showed their practical value in helping to pinpoint these problems in students' programming assignments (see Section 8).

5.1.1. Program unrolling

CFAULTS starts the unrolling process by expanding the faulty program using the set of failing tests from the test suite. In this context, an unrolled program signifies the original program expanded m times (m program scopes), where m denotes the number of failed test cases. An unrolled program encodes the execution of all failing tests within the program, along with their corresponding inputs and specifications (assertions).

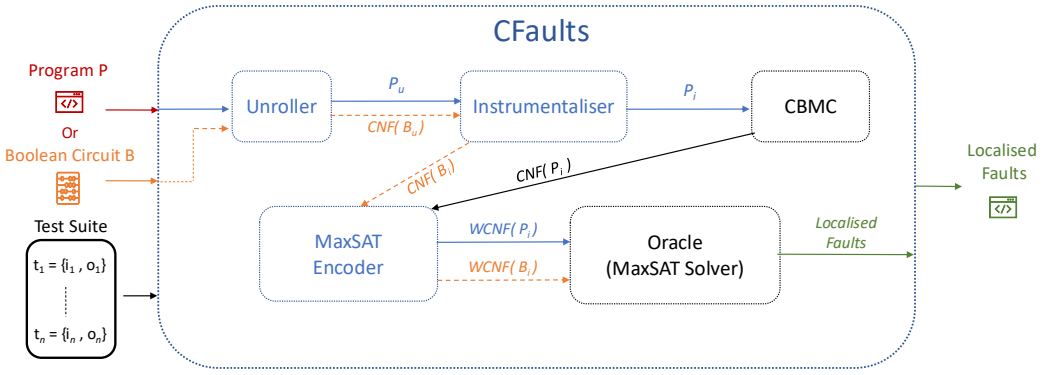


Figure 1: Overview of CFaults.

The unrolling process encompasses three primary steps. Initially, CFaults generates *fresh variables* and functions for each of the m program scopes, ensuring each scope possesses unique variables and functions. Subsequently, CFaults establishes variables representing the inputs and outputs for each program scope corresponding to the failing tests.

Input operations, such as `scanf`, undergo translation into read accesses to arrays corresponding to the inputs, while output operations, such as `printf`, are replaced by write operations into arrays representing the program's output. Every exit point of the program (e.g., a `return` statement in the `main` function) is replaced with a `goto` statement directing the program flow to the next failing test's scope. Furthermore, to avoid the C compiler's default zero-initialisation, all uninitialised global variables in the program are explicitly initialised to nondeterministic values.

Lastly, at the end of the unrolled program, CFaults embeds an assertion capturing all the specifications of the failing tests. Thus, the unrolled program encapsulates the execution of all failing tests within a single program.

Listing 2 exhibits a program segment generated through the unrolling process applied to Listing 1. CFaults establishes global variables to represent the inputs and outputs of each failing test (lines 1–3, Listing 2). For the sake of simplicity, the depicted listing illustrates solely the initial scope corresponding to test 0 from the test suite outlined in Table 1. Distinct variables are introduced for each failing test. Furthermore, the `scanf` function call is substituted with input array operations (lines 8–10), while the `printf` calls are replaced with CFaults' print functions, akin to `sprintf` functions, which direct output to a buffer. Lastly, the unrolled program concludes with

Listing 2: The program from Listing 1 after being subjected to CFaults' unrolling process, using the test suite presented in Table 1. For simplicity, only the initial scope corresponding to test t_0 is displayed. The scopes `scope_1` and `scope_2` associated with failing tests t_1 and t_2 are omitted.

```
1  float _input_f0[3] = {1, 2, 3};
2  char _out_0[2] = "3";
3  int _ioff_f0 = 0, _ooff_0 = 0;
4  // ... inputs and outputs for the other tests
5  int main(){
6    scope_0:{
7      int f_0, s_0, t_0;
8      f_0 = _input_f0[_ioff_f0++];
9      s_0 = _input_f0[_ioff_f0++];
10     t_0 = _input_f0[_ioff_f0++];
11     if ((f_0 < s_0) && (f_0 >= t_0))
12         _ooff_0 = printInt(_out_0, _ooff_0, f_0);
13     if ((f_0 > s_0) && (s_0 <= t_0))
14         _ooff_0 = printInt(_out_0, _ooff_0, s_0);
15     if ((f_0 > t_0) && (s_0 > t_0))
16         _ooff_0 = printInt(_out_0, _ooff_0, t_0);
17     goto scope_1;
18   }
19   // ... scope_1 and scope_2
20   final_step:
21   assert(strcmp(_out_0, "3") != 0 || // other assertions);
22 }
```

an assertion representing the disjunction of the negation of all failing test assertions. For instance, suppose there are m failing tests, where A_i denotes the assertion of test t_i. In this scenario, CFaults injects the following assertion into the program: $\neg A_1 \vee \cdots \vee \neg A_m$.

5.1.2. Program Intrumentalisation

After integrating all possible executions and assertions from failing tests during the unrolling step, CFaults proceeds to instrumentalise the unrolled C program by introducing *relaxation variables* for each program component (i.e., statement, instruction). These relaxation variables correspond to the healthy variables explained in Section 4. Thus, each relaxation variable activates (or deactivates) the program component being relaxed when assigned to `true` (or `false`) respectively. CFaults ensures that there are no conflicts between the names of the relaxation variables and the names of the program's original variables. For this step, CFaults needs to receive a maximum number of iterations that the program should be unwound.

The relaxation process introduces relaxation variables that deactivate or activate program components. This process involves four distinct relaxation rules for: (1) conditions of `if`-statements, (2) expression lists (e.g., an expression list executed at the beginning of a for-loop), (3) loop conditions,

Listing 3: Program statements.

```
1   int i;
2   int n;
3   int s;
4
5   s = 0;
6   n = _input_f0[_ioff_f0++];
7
8   if (n == 0)
9       return 0;
10
11  for (i=1; i < n; i++){
12      s = s + i;
13  }
```

Listing 4: Program statements relaxed.

```
1   //main scope
2   bool _rv1, _rv2, _rv3, _rv5;
3   bool _rv6[UNWIND],..., _rv9[UNWIND];
4   int _los; // loop1 offset
5
6   //test scope
7   bool _ev4, _ev7[UNWIND];
8   int i,n,s;
9   _los=1;
10
11  if (_rv1) s = 0;
12  if (_rv2) n = _input_f0[_ioff_f0++];
13
14  if ( _rv3 ? (n == 0) : _ev4)
15      return 0;
16
17  for (_rv5 ? (i = 1) : 1;
18       _rv6[_los] ? (i<n) : _ev7[_los];
19       _rv9[_los] ? i++ : 1, _los++){
20      if (_rv8[_los]) s = s + i;
21  }
```

and (4) other program statements.

Example 8. *Listings 3 shows a code snippet that sums all numbers between* 1 *and* `n`*. Listings 4 depicts the same program statements after undergoing relaxation by* CFaults*. For the sake of simplicity, all relaxation variable and offset names were simplified.*

In more detail, the rule for relaxing a general program statement is to envelop the statement with an `if`-statement, whose condition is a relaxation variable. For example, consider lines 5 and 6 in the program on Listings 3. These lines are relaxed by CFaults using relaxation variables `_rv1` and `_rv2` respectively, appearing as lines 11 and 12 on Listings 4.

Furthermore, when relaxing `if`-statements, the statements inside the `then` and `else` blocks adhere to the previously explained relaxation rule. However, the conditions of `if`-statements are relaxed using a ternary operator, as shown in line 14 of Listings 4. Note that if the relaxation variable is assigned `true`, then the original `if` condition is executed. Otherwise, a different relaxation variable (e.g., `_ev4` in Listings 4) determines whether the program execution enters the `then`-block or the `else`-block (if one exists). These relaxation variables (`else`*'s relaxation variables*) are local to each failing test scope and enable different tests to determine whether to enter the `then` or `else`-block.

When handling expression lists, CFaults adopts a comparable strategy to that of generic program statements, enclosing each expression within a

ternary operator instead of an `if`-statement. If the program component is deactivated, the expression is replaced by `1`. For example, the initialisation of variable `i` in line 11 of Listings 3 is relaxed into the ternary operation in line 17 of Listings 4.

Lastly, all relaxation variables inside a loop are Boolean vectors to relax statements within the loop. Each entry of these vectors relaxes the loop's statements for a given iteration. The maximum number of iterations of the loops is defined by the CFaults user. CFaults follows a similar approach for inner loops, creating arrays of arrays. Thus, for simple program statements within a loop, CFaults encapsulates them with `if`-statements, with the relaxation variables indexed to the iteration number. Line 20 of Listings 4 illustrates a relaxed statement inside a loop. The loop's condition is relaxed by implication of the relaxation variable, as demonstrated in line 18 of Listings 4. Furthermore, each loop has its own offsets to index relaxation variables. These offsets are initialised just before the loop and incremented at the end of each iteration (e.g., line 19 in Listing 4).

We would like to point out that our relaxation of loop conditions (see line 18 in Listing 4) differs from the approach adopted in the previous version of this work [5], which relaxed the loop condition as follows:

```
(!_rv6[_los] || (i<n))
```

However, this relaxation strategy was incomplete, as it did not account for non-terminating loops, i.e., scenarios where the loop condition is always assigned to true. In contrast, our current approach mirrors the relaxation method applied to `if`-statement conditions, and we introduce `else` *relaxation variables* to control the execution path when the original condition is deactivated. This ensures that the analysis accurately captures all control flow behaviours, including potentially non-terminating loops.

When handling auxiliary functions, CFaults declares the relaxation variables needed in the main scope of the program and passes these variables as parameters. Hence, CFaults ensures that the same variables are used throughout the auxiliary functions' calls.

Listing 5 depicts the program resulting from the instrumentalisation process of Listing 2 performed by CFaults. The same program components (i.e., statements, instructions) across different failing test scopes are assigned the same relaxation variable declared in the main scope. Consequently, if a relaxation variable is set to `false`, the corresponding program component is deactivated across all test executions. Additionally, the relaxation variables

Listing 5: Instrumentalised program.

```
1  //global vars
2  int main(){
3    bool _rv1, _rv2, ..., _rv12;
4    scope_0:{
5      bool _ev5, _ev8, _ev11;
6      int f_0, s_0, t_0;
7      if (_rv1) f_0 = _input_f0[_ioff_f0++];
8      if (_rv2) s_0 = _input_f0[_ioff_f0++];
9      if (_rv3) t_0 = _input_f0[_ioff_f0++];
10     if (_rv4 ? ((f_0 < s_0) && (f_0 >= t_0)) : _ev5 ){
11         if (_rv6) _ooff_0 = printInt(_out_0, _ooff_0, f_0);
12     }
13     if (_rv7 ? ((f_0 > s_0) && (s_0 <= t_0)) : _ev8 ){
14         if (_rv9) _ooff_0 = printInt(_out_0, _ooff_0, s_0);
15     }
16     if (_rv10? ((f_0 > t_0) && (s_0 > t_0)) : _ev11 ){
17         if (_rv12) _ooff_0 = printInt(_out_0, _ooff_0, t_0);
18     }
19     goto scope_1;
20   }
21   // scope_1 and scope_2
22   final_step:
23   assert(strcmp(_out_0, "3") != 0 || ... // other assertions);
24 }
```

are left uninitialised, allowing CFaults to determine the minimum number of faulty components requiring deactivation. Note that relaxation variables are not declared as global variables but as local variables within the `main` scope. This is to prevent the C compiler from automatically initialising all these variables to `false`.

5.1.3. CBMC

After unrolling and instrumentalising the C program, CFaults invokes CBMC, a bounded model checker for C [19]. First, CFaults invokes CBMC with memory-safety checks enabled (e.g., `-bounds-check`, `-pointer-check`) to detect inconsistencies in the unrolled, instrumented buggy program that our MBD approach cannot capture, as modelling these violations would cause the SAT solver to consider assignments that do not correspond to valid C states. If CBMC reports such issues, CFaults terminates immediately and returns an informative error message to alert the user to invalid memory accesses. Next, CFaults uses CBMC to transform the unrolled and relaxed program into *Static Single Assignment (SSA)* form [39], an intermediate representation ensuring that variables are assigned values only once and are defined before use. SSA achieves this by converting existing variables into multiple versions, each uniquely representing an assign-

ment. Next, CBMC translates the SSA representation into a CNF formula, which represents the trace formula of the program. During the CNF formula generation, CBMC negates the program's assertion ($\neg(\neg A_1 \vee \cdots \vee \neg A_m)$) to compute a counterexample. Moreover, the CNF formula, ϕ, encodes each failing test's input (I_i), assertion (A_i), and all execution paths of the unrolled and relaxed incorrect program encoded by the trace formula (P_i), i.e., $\phi = (I_1 \wedge \ldots \wedge I_m) \wedge P_i \wedge (A_1 \wedge \cdots \wedge A_m)$. Thus, if ϕ is SAT, an assignment exists that activates or deactivates each relaxation variable and makes all failing test assertions true. Hence, each satisfiable assignment is a diagnosis of the C program, considering all failing tests.

5.1.4. MaxSAT Encoder

Let ϕ denote the CNF formula generated by CBMC in the previous step. Next, CFAULTS generates a weighted partial MaxSAT formula $(\mathcal{H}, \mathcal{S})$ to maximise the satisfaction of relaxation variables in the program, aiming to minimise the necessary code alterations. The set of hard clauses is defined by CBMC's CNF formula (i.e., $\mathcal{H} = \phi$), while the soft clauses consist of unit clauses representing relaxation variables used to instrument the C program, expressed as $\mathcal{S} = \bigwedge_{c \in \mathcal{C}} (rv_c)$.

Additionally, CFAULTS assigns hierarchical weights to relaxation variables according to the height of their sub-ASTs (*abstract syntax trees* [40]). For example, for an `if` statement without an `else` branch, the relaxation variable associated with the condition receives a weight equal to the sum of the weights of the relaxation variables in the `then` block. Furthermore, to prioritise identifying faults in the program logic over handling input/output issues, I/O statements (e.g., `scanf`, `printf`) are given substantially higher weights than other statements. The hierarchical weighting scheme is optional and can be enabled or configured via CFAULTS command-line options.

Moreover, CFAULTS enumerates all *optimal* MaxSAT solutions to obtain every subset-minimal diagnosis of minimum total weight, since multiple optimal solutions (with identical optimum cost) can correspond to different sets of relaxed program statements.

5.1.5. Oracle

CFAULTS invokes a MaxSAT solver to determine the program's diagnoses with minimum weight. Note that all these diagnoses are also subset-minimal, aligning with the principles of Model-Based Diagnosis (MBD) theory. By consolidating all failing tests into a unified, unrolled, and instrumentalised

program, the MaxSAT solution identifies the minimum subset of statements requiring removal to fulfil the assertions of all failing tests.

5.2. CFaults sees Fault in Circuits

As indicated by the orange dashed arrows in Figure 1, when CFaults receives a Boolean circuit `B` and a test suite as input, it first unrolls the circuit, then instruments the unrolled circuit, and finally encodes it into MaxSAT before invoking a MaxSAT oracle.

5.2.1. Circuit unrolling

CFaults begins the unrolling process by expanding the faulty Boolean circuit using the set of failing input-output observations from the test suite. The unrolled circuit B_u encodes the execution of all failing observations within a single circuit, by encoding their corresponding inputs and outputs. During unrolling, CFaults generates *fresh Boolean variables* for each of the m circuit copies, one for each failing observation, ensuring that every copy has its own unique variables corresponding to its inputs, outputs, and internal gates. Thus, an unrolled circuit B_u represents the original circuit `B` replicated m times, where m is the number of failing test cases, i.e., $B_u \leftrightarrow (i_1 \wedge B_1 \wedge o_1) \wedge (i_2 \wedge B_2 \wedge o_2) \wedge \cdots \wedge (i_m \wedge B_m \wedge o_m)$, where i_k and o_k correspond to the input-output observation of circuit B_k.

5.2.2. Circuit instrumentalisation

During this step, CFaults relaxes each inner gate of the unrolled circuit B_u, using relaxation variables (i.e., the *healthy variables* described in Section 4). Each relaxation variable controls the activation of its corresponding circuit component, enabling it when set to `true` and disabling it when set to `false`. Moreover, identical circuit components (i.e., inner gates) across different circuit copies from different observations share the same relaxation variable. As a result, if a relaxation variable is set to `false`, the corresponding inner gate is deactivated simultaneously across all failing observations. For example, let g_j denote the Boolean variable corresponding to inner gate j. Then, CFaults adds the clause $rv_j \implies g_{jk}$ for each observation k, generating the unrolled and relaxed Boolean circuit B_i.

5.2.3. MaxSAT Encoder

To encode this MBD problem into MaxSAT, CFaults generates an unweighted partial MaxSAT formula $(\mathcal{H}, \mathcal{S})$ that maximises the satisfaction of

the relaxation variables, thereby minimising the required gate modifications. The set of hard clauses is given by the unrolled and instrumented circuit B_i, as described in the previous sections (i.e., $\mathcal{H} = B_i$), while the soft clauses consist of unit clauses representing the relaxation variables used to instrument the Boolean circuit, expressed as $\mathcal{S} = \bigwedge_{c \in \mathcal{C}}(rv_c)$, where $\mathcal{C}$ denotes the set of circuit gates.

5.2.4. Oracle

CFaults calls a MaxSAT solver to find the circuit's smallest set of faulty components, aligning with the principles of MBD theory with multiple observations presented in Section 4. By consolidating all failing input-output examples into a unified and unrolled circuit, the MaxSAT solution identifies the minimum subset of components requiring removal to fulfil all failing input-output observations.

6. Experimental Results

All of the experiments were conducted on an Intel(R) Xeon(R) Silver computer with 4110 CPUs @ 2.10GHz running Linux Debian 12, using a memory limit of 32 GB and a timeout of 3600s, for each buggy program, and 1800s for each buggy Boolean circuit.

Section 6.1 presents the experiments of running CFaults and other FBFL methods on benchmarks of C programs. Section 6.2 presents the results of using CFaults and HSD [9] to localise bugs on Boolean circuits. For the MaxSAT oracle, RC2Stratified [41] from the `PySAT` toolkit [42] was used.

6.1. C Software

Benchmarks. CFaults has been evaluated using two distinct benchmarks of C programs: TCAS [20] and C-Pack-IPAs [21].

TCAS stands out as a well-known program benchmark extensively studied in the fault localisation literature [10, 11]. This benchmark comprises a C program from Siemens and 41 versions with intentionally introduced faults, with known positions and types of these faults. Conversely, C-Pack-IPAs is a collection of student programs gathered from an introductory programming course. For this evaluation, the entire C-Pack-IPAs benchmark was used, consisting of twenty-five programming assignments and a total of 1431 faulty programs drawn from three laboratory classes. These programs cover a range of topics, including integers and input–output operations, loops and character

manipulation, and vector processing. C-Pack-IPAs has proven successful in evaluating various works across program analysis tasks [43, 44, 45, 46].

CFaults uses `pycparser` [47] for unrolling and instrumentalising C programs. Additionally, CBMC version 5.11 is used to encode C programs into CNF formulae.

Fault Localisation Methods. Since the source code of BugAssist and SNIPER is either unavailable or no longer maintained (resulting in compilation and linking issues), prototypes of their algorithms were implemented. It is worth noting that the original version of SNIPER could only analyse programs that utilised a subset of ANSI-C, lacked support for loops and recursion, and could only partially handle global variables, arrays, and pointers. Moreover, we have also implemented our prototype for HSD [9] to localise faulty statements in ANSI-C software, both with and without core minimisation (HSD-CM and HSD, respectively), since the original algorithm was only implemented for Boolean circuits, and not for C software. Thus, in this work, BugAssist, SNIPER and HSD can handle ANSI-C programs, as their algorithms are built on top of CFaults's unroller and instrumentaliser modules.

All FBFL algorithms evaluated consistently generate diagnoses that are consistent with (5), indicating that all proposed diagnoses undergo validation by CBMC once the algorithm provides a diagnosis. However, this validation primarily serves to verify diagnoses generated by BugAssist, as it has the capability to produce diagnoses that may not align with all failing test cases. In contrast, both CFaults' MaxSAT solution, and HSD hitting set approach, by definition, align with all observations. SNIPER's aggregation method (Cartesian product) produces only valid diagnoses, although they may not always be subset-minimal. When considering BugAssist, as presented in Algorithm 1, we iterate through all computed diagnoses based on BugAssist's voting score, until we identify one diagnosis that is consistent with all observations, i.e., consistent with (5).

Hierarchical Weights. For both C benchmark, TCAS and C-Pack-IPAs, we evaluate all FBFL approaches *with* and *without* hierarchical weights on the soft clauses (relaxation variables; see Section 5.1.4). As the results are similar, we report only the weighted case here; unweighted results appear in Appendix A. As explained in Section 5.1.4, hierarchical weights guide the search toward a minimum set of buggy statements by accounting for the statement's

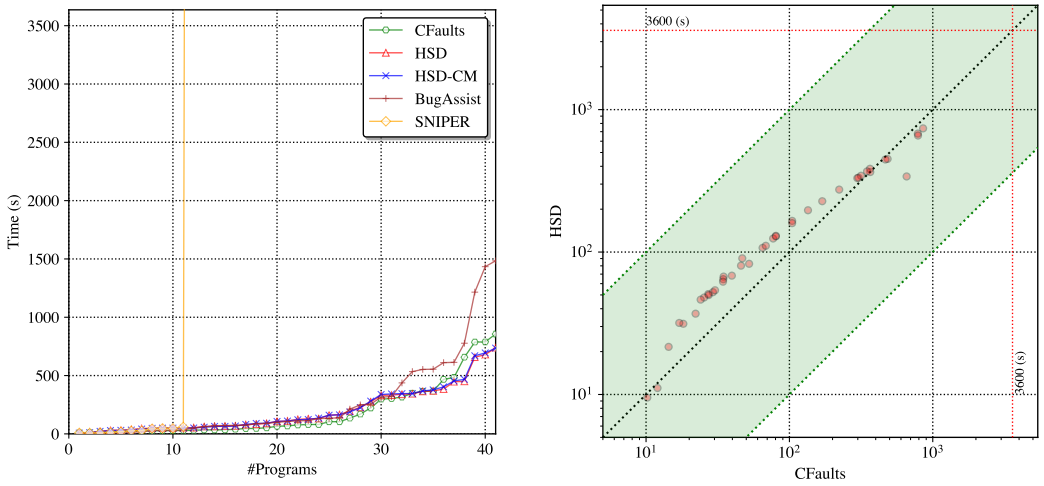
Benchmark: **TCAS**			
	Valid Diagnosis	**Memouts**	**Timeouts**
BugAssist	**41 (100.0%)**	0 (0.0%)	0 (0.0%)
SNIPER	11 (26.83%)	**30 (73.17%)**	0 (0.0%)
HSD	**41 (100.0%)**	0 (0.0%)	0 (0.0%)
HSD-CM	**41 (100.0%)**	0 (0.0%)	0 (0.0%)
CFaults	**41 (100.0%)**	0 (0.0%)	0 (0.0%)

Table 3: Fault localisation results on TCAS [20] benchmark.

sub-AST (abstract syntax tree). Furthermore, since TCAS comprises multiple versions of the same program, the initialisation of program variables can be treated as hard constraints (see Section 4), since these initialisations are never faulty in this benchmark. In contrast, C-Pack-IPAs contains student programs where faults may occur anywhere, including in variable initialisation; accordingly, we relax all executable statements, excluding variable and function declarations, across all FBFL approaches.

Static Analysers. Before executing any FBFL algorithms, every C program undergoes a series of static checks, as explained in Section 5. Specifically, prior to unrolling and instrumenting the C programs, the static analysis tools cppcheck [36] and clang-tidy [37] are invoked to detect issues such as uninitialised variables, errors like division by zero, and outputs differing only in white space. If any issues are detected, the fault localisation procedure is immediately terminated, and the statically identified faults are returned to the user. Otherwise, the given buggy program proceeds to unrolling and instrumentation. The entire set of programs in the TCAS benchmark was analysed in this way, and no static issues were reported. By contrast, the static analysers detected faults such as uninitialised variables in 749 programs from the C-Pack-IPAs benchmark (approximately 52%). Accordingly, in this section we present FBFL results only for the remaining 48% of C programs in this benchmark.

TCAS. Table 3 provides an overview of the results obtained using BugAssist, SNIPER, HSD, HSD-CM, and CFaults on TCAS. Entries highlighted in bold correspond to the highest value per column. All FBFL methods, except SNIPER, find valid diagnoses for every instance in TCAS. The TCAS program comprises approximately 180 lines of code and has a maxi-



(a) Time performance of all FBFL approaches.

(b) Time comparison between CFaults and HSD.

Figure 2: Total CPU time comparison between BugAssist, SNIPER, HSD and CFaults on the TCAS [20] benchmark.

mum of 131 failing tests for each program. As shown in Table 3, this leads SNIPER to reach the memory limit of 32GB for 73% of the programs when aggregating the sets of MCSes computed for each failing test.

Figure 2a presents a *cactus plot* showing the CPU time required for localising faults in each program (y-axis) against the number of programs with faults successfully localised (x-axis), using all FBFL algorithms on TCAS. Overall, CFaults demonstrates faster performance than BugAssist and SNIPER, and is generally slightly faster than HSD. The performance of SNIPER can be explained by its high memout rate on this benchmark (see Table 3).

Furthermore, Figure 2b shows a *scatter plot* comparing the CPU times of CFaults and HSD on TCAS. Each point in the plot corresponds to a faulty program, where the x-value (respectively, y-value) denotes the time spent by CFaults (respectively, HSD) in localising faults in that program. Points lying above the diagonal indicate programs where HSD required more CPU time than CFaults to generate a valid diagnosis. As shown, CFaults is generally faster than HSD on fault localisation for the TCAS benchmark.

Moreover, although BugAssist localised faults in all programs, it yielded a non-optimal diagnosis in 2.4% of cases. Additionally, SNIPER enumerated redundant diagnoses in 27% of the programs in which it successfully

Benchmark: **C-Pack-IPAs**			
	Valid Diagnosis	**Memouts**	**Timeouts**
BugAssist	265 (38.86%)	5 (0.73%)	412 (60.41%)
SNIPER	234 (34.31%)	**21 (3.08%)**	**427 (62.61%)**
HSD	375 (54.99%)	12 (1.76%)	295 (43.26%)
HSD-CM	371 (54.40%)	13 (1.91%)	298 (43.70%)
CFaults	**480 (70.38%)**	14 (2.05%)	188 (27.57%)

Table 4: Fault localisation results on C-Pack-IPAs [21] benchmark.

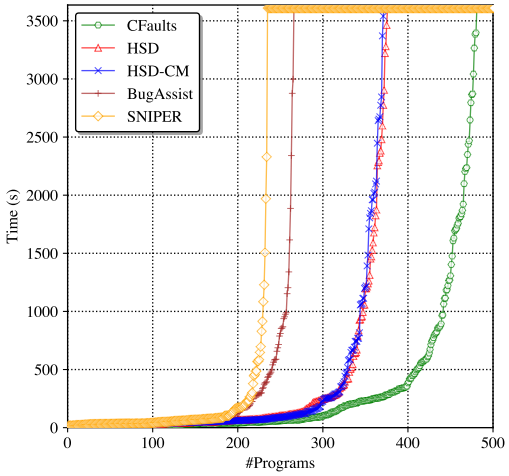
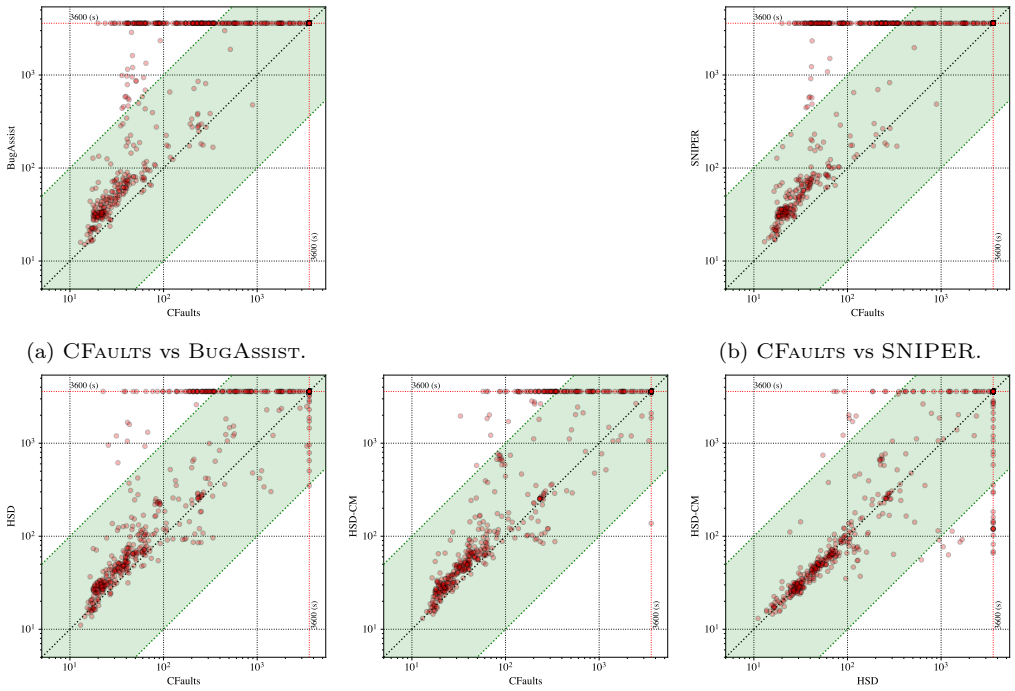


Figure 3: Total CPU time performance of all FBFL approaches on C-Pack-IPAs.

localised faults.

C-Pack-IPAs. Regarding the C-Pack-IPAs benchmark, Table 4 provides an overview of the results obtained using BugAssist, SNIPER, HSD, HSD-CM, and CFaults on this benchmark. Entries highlighted in bold correspond to the highest value per column. Figure 3 presents a cactus plot of the total CPU time spent by all FBFL algorithms on C-Pack-IPAs. A similar trend can be observed here: CFaults is faster at localising bugs in C software and is able to localise significantly more bugs than all other FBFL approaches. These results are consistent with Table 4, which shows that CFaults localises faults in 31% more programs than BugAssist, 36% more programs than SNIPER, and 15% more programs than HSD when considering its best-performing configuration on C-Pack-IPAs (i.e., with-



(a) CFaults vs BugAssist.

(b) CFaults vs SNIPER.

(c) CFaults vs HSD.

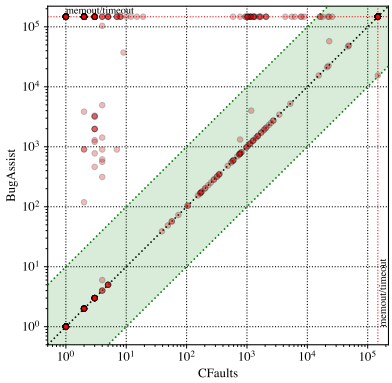
(d) CFaults vs HSD-CM.

(e) HSD vs HSD-CM.

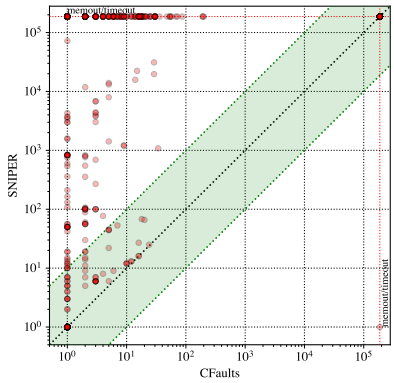
Figure 4: CPU time comparison between BugAssist, SNIPER, HSD, HSD with core minimisation (HSD-CM) and CFaults on C-Pack-IPAs.

out core minimisation). Moreover, Figure 4 presents several scatter plots comparing the total CPU time of CFaults against each of the other localisation methods, further illustrating that, in general, CFaults is faster than the other FBFL approaches when localising faults in the same C programs.

Furthermore, Figure 4e shows that the use of unsatisfiable core minimisation in HSD (see Section 4.1) yields a performance profile similar to that obtained without minimisation. For all programs in C-Pack-IPAs where HSD successfully localised faults, the minimum, average, and maximum numbers of algorithm iterations (i.e., number of cores enumerated) with core minimisation were 1, 7, and 135, respectively, compared to 1, 105, and 2566 without core minimisation. In terms of CPU usage, HSD-CM devoted on average around 98% of its total computation time to core minimisation in programs where faults were localised within the time limit. For programs where the time limit was exceeded, HSD-CM spent on average about 93% of the computation time minimising unsatisfiable cores. Thus, while core minimisation



(a) MaxSAT costs of BugAssist's and CFaults' diagnoses.



(b) #Diagnoses generated by SNIPER and CFaults.

Figure 5: Comparison between BugAssist's, SNIPER's and CFaults' diagnoses.

enables HSD to reach minimal diagnoses in fewer iterations, it comes at a cost, as shown in Table 4, the number of faults localised decreases by 0.5%, memouts increase by 0.15%, and timeouts increase by around 0.5%.

Finally, Figure 5a illustrates a scatter plot comparing the diagnoses' costs of the MaxSAT formulae (i.e., total weight of unsatisfied soft clauses) achieved by CFaults (x-axis) against BugAssist (y-axis) on C-Pack-IPAs. This plot shows that BugAssist fails to provide an optimal diagnosis in 9% of cases. The other FBFL approaches (i.e., SNIPER, HSD, and CFaults) always return a minimal diagnosis of optimal cost. However, unlike CFaults and HSD, SNIPER typically reaches the optimum only after enumerating a large number of redundant (non-optimal) diagnoses. Figure 5b depicts a scatter plot comparing the number of diagnoses enumerated by CFaults (x-axis) against SNIPER (y-axis). This figure shows that while CFaults and HSD enumerate all optimal solutions of the weighted MaxSAT formulation, SNIPER produces far more diagnoses overall. In particular, SNIPER enumerates redundant diagnoses in 55% of the programs in which it successfully localised faults, suggesting that its enumeration procedure does not prevent redundant diagnoses. The number of such redundant diagnoses is much larger than the subset-minimal diagnoses generated by CFaults and HSD. Figure 5b illustrates that in some instances, SNIPER may enumerate up to 100K diagnoses, whereas CFaults generates less than 10 diagnoses.

Thus, although CFaults incorporates all failing test cases into a single

Benchmark: **ISCAS85**			
#Observations	**CFaults**	**HSD**	**HSD-CM**
10	2011 (81.62%)	**2135 (86.65%)**	2110 (85.63%)
20	1865 (76.12%)	**2004 (81.80%)**	1983 (80.94%)
30	1720 (71.70%)	**1878 (78.28%)**	1860 (77.53%)
40	1709 (70.82%)	**1886 (78.16%)**	1881 (77.95%)
50	1664 (68.73%)	1832 (75.67%)	**1844 (76.17%)**
All	8969 (73.84%)	**9735 (80.14%)**	9678 (79.67%)

Table 5: Number of successfully localised minimum diagnoses by HSD (with and without core minimisation) and CFaults on the augmented ISCAS85 [22] benchmark. Time limit: 1800s, the memory limit was not reached for any instance.

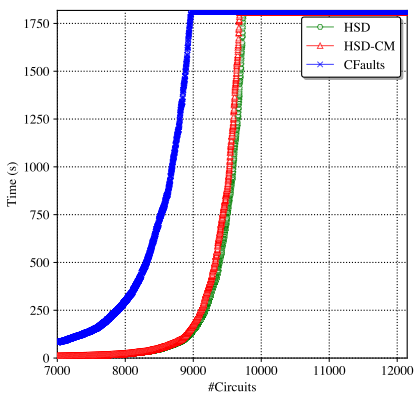
MaxSAT formula and must compute all MaxSAT solutions of that formula, it remains faster and localises more faults on both C benchmarks, TCAS and C-Pack-IPAs, than the other FBFL approaches (i.e., BugAssist, SNIPER, and HSD), which divide the localisation process into multiple MaxSAT calls.

6.2. Boolean Circuits

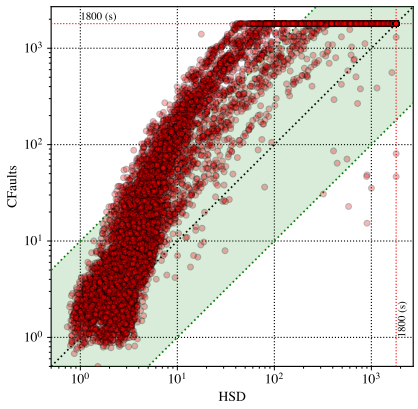
Benchmark. For the Boolean circuit evaluation, we compared CFaults with the HSD algorithm [9], both with and without core minimisation (HSD-CM and HSD, respectively), on the widely studied ISCAS85 benchmark [22]. To extend this benchmark, we injected both single and multiple faults using the publicly available HSD [9] code. In particular, we varied the number of failing observations (10, 20, 30, 40, and 50) and the number of injected faults (1, 2, 3, 4, 5, 10, 20, 30, 40, and 50). Thus, the benchmark used in this experiment comprises 12147 buggy Boolean circuits.

For the ISCAS85 benchmark, we evaluate all FBFL approaches *without* hierarchical weights on the soft clauses (i.e., relaxation variables). Such weighting is primarily useful for C programs, where disabling a statement can alter control flow. By contrast, in Boolean circuits, masking a gate does not block the evaluation of downstream gates, so hierarchical weights provide no additional guidance. Hence, for this benchmark we solve an *unweighted* MaxSAT problem, assigning the same cost to all gates.

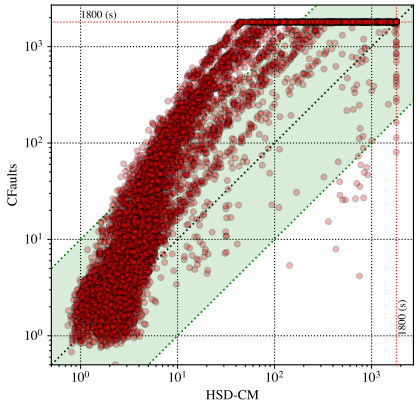
Table 5 reports the number of minimum diagnoses successfully localised by HSD, HSD-CM and CFaults on the augmented ISCAS85 benchmark, under varying numbers of failing observations. Entries highlighted in bold correspond to the highest value per number of observations. These results



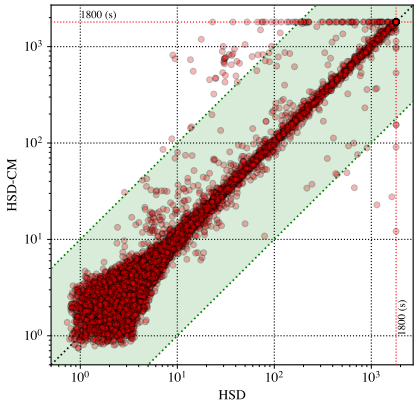
(a) CPU time performance of HSD and CFaults.



(b) Time comparison between CFaults and HSD.



(c) Time comparison between CFaults and HSD with core minimisation.



(d) Time comparison between HSD with and without core minimisation.

Figure 6: CPU time comparison between HSD and CFaults on the ISCAS85 [22].

indicate that CFaults remains highly competitive with HSD when the number of failing observations is small (10 or 20), localising faults in only 4–5% fewer circuits than HSD, depending on whether core minimisation is applied. For larger numbers of failing observations (30, 40, 50), the gap increases modestly to about 6%. Aggregating over all circuits (i.e., without grouping by observation count), the gap between CFaults and HSD is around 6%. Notably, HSD performs almost identically with and without core minimisation, differing by at most 1%.

Figure 6 complements these results by comparing the CPU time performance of CFaults and HSD. In particular, Figure 6a presents a cactus

plot depicting CPU time (y-axis) against the number of circuits successfully localised (x-axis), using CFaults, HSD, and HSD-CM on the augmented ISCAS85 benchmark. The results confirm that HSD achieves faster performance overall and localises faults in a larger number of circuits, in line with the findings of Table 5.

Additional insights are provided by the scatter plots in Figures 6b, 6c, and 6d, which compare the CPU times of CFaults and HSD. For example, in Figure 6b, each point corresponds to a faulty circuit, with the x-value (respectively, y-value) representing the time taken by HSD (respectively, CFaults) to localise faults in that circuit. Points lying above the diagonal denote circuits where CFaults required more time than HSD to compute a valid diagnosis. These plots clearly demonstrate that HSD, both with and without core minimisation, consistently outperforms CFaults in terms of CPU time.

Moreover, Figure 6c shows that the use of unsatisfiable core minimisation in HSD (see Section 4.1) results in performance that is almost indistinguishable from that obtained without minimisation. For all circuits where HSD successfully localised faults, the minimum, average, and maximum numbers of cores enumerated without core minimisation were 1, 101, and 33532, respectively, compared to 1, 50, and 6403 with minimisation. The average time spent by HSD on core minimisation in this benchmark was less than 1%. For circuits where CFaults failed to localise faults within the time limit, the corresponding numbers of HSD iterations were 4, 543, and 33532 without core minimisation, and 4, 155, and 6012 with minimisation.

In summary, on the augmented ISCAS85 benchmark with multiple injected faults and varying numbers of failing observations, CFaults is generally slower and succeeds in localising faults in fewer Boolean circuits overall than HSD. Nevertheless, CFaults remains competitive, localising faults in only 6% fewer circuits than HSD.

6.3. Discussion

In the evaluation on buggy C software, using the TCAS and C-Pack-IPAs benchmarks, CFaults demonstrated clear advantages over the competing FBFL methods. Even though CFaults encodes all failing test cases into a single MaxSAT formula, it consistently outperformed the other approaches. In particular, CFaults was both faster and able to localise more faults on C-Pack-IPAs, identifying faults in 31% more programs than BugAssist, 36% more programs than SNIPER, and 15% more programs than

HSD, all of which decompose the fault localisation task into multiple independent MaxSAT problems.

By contrast, in the evaluation on Boolean circuits with multiple injected faults and varying numbers of failing observations, the performance trends differ. In this setting, CFaults is generally slower than HSD and localises faults in fewer circuits overall. However, the difference is modest since CFaults localises faults in only 6% fewer circuits than HSD, indicating that it remains a competitive approach.

Taken together, these results suggest that among the FBFL approaches considered, CFaults is the fastest and most reliable method for C software, whereas for Boolean circuits it remains competitive, but HSD continues to represent the state of the art. One factor helps to explain this difference in performance. Fault localisation in C programs and Boolean circuits constitutes two structurally distinct problems, leading to MaxSAT encodings with significantly different formulae structures. This distinction has a substantial influence on solver efficiency, contributing to the observed disparity between the two domains.

7. Threats to Validity

Faulty code not being executed. Our approach requires that faulty statements are *executed* by at least one failing test case; statements that are never executed cannot be diagnosed. Consequently, insufficient input coverage can yield false negatives (i.e., missed diagnoses). We partially mitigate this with multiple failing observations and lightweight static analysis.

Bounded model checking (BMC). We rely on bounded encodings of programs. Loop unrolling and depth bounds can exclude feasible behaviours, which may render otherwise valid diagnoses unreachable under a given bound. Moreover, when bounds truncate iterations, diagnoses that hold for the bounded model may not generalise to the original program. We choose bounds conservatively and report them alongside results, but incompleteness due to bounding is an inherent limitation.

Pointers, arrays, and undefined behaviour. As explained in Section 5.1, CFaults relies on static analysers such as, cppcheck [36] and clang-tidy [37], to detect issues such as uninitialised variables and potential runtime errors (e.g., division by zero). If any such issues are detected, CFaults terminates immediately and reports them to the user. Furthermore, CFaults

enables CBMC's memory-safety checks (e.g., `-bounds-check`, `-pointer-check`) to detect invalid memory operations, including out-of-bounds accesses and unsafe pointer arithmetic. However, CFaults inherits its CBMC's specification and encoding of ANSI C: behaviours deemed well defined by CBMC's memory model are treated as valid by our encoding, whereas violations that CBMC does not flag may escape detection. Clarifying, and, where possible, tightening, the alignment between CBMC's configuration and the intended C semantics for pointers and arrays is left for future work.

External validity. Our evaluation uses benchmark and student programs that compile successfully and include explicit specifications (assertions or expected outputs). Generalising to larger, system-scale codebases; programs with extensive I/O, concurrency, or libraries with complex contracts; or to specifications expressed in richer formalisms may require additional engineering and could alter the results observed in this study.

8. CFaults: Use Cases

This section presents several use cases of fault localisation in C software where CFaults has been successfully applied.

Providing feedback in programming tasks. In Spring 2024, we evaluated the use of CFaults to provide automated error feedback to students in a first-year undergraduate programming course in C [38]. CFaults was integrated with GitSEED [38], a GitLab-backed automated assessment tool for software engineering and programming education, which served as the platform for assessing laboratory work and projects in the course. Among the findings of this study, 69% of the surveyed students reported that the "hints" (i.e., localised faults) generated by CFaults were helpful. These hints provided valuable guidance by pointing students towards potential errors in their code and promoting a deeper understanding of programming concepts through self-directed correction. Overall, this study [38] highlighted the positive impact of CFaults as a feedback mechanism, demonstrating its value as a comprehensive educational tool in programming courses.

Guiding LLMs on automated program repair. CFaults has also been integrated with Large Language Models (LLMs), via zero-shot learning, to enhance automated program repair for C programming assignments [44, 46]. In this setting, CFaults identifies the buggy parts of a C program and

presents the LLM with a program sketch in which these faulty statements are removed. This hybrid approach follows a Counterexample Guided Inductive Synthesis (CEGIS) loop [48] to iteratively refine the program. The LLM is then asked to synthesise the missing parts, which are subsequently checked against a test suite. If the synthesised program is incorrect, a counterexample from the test suite is provided to the LLM to guide a revised synthesis. This method enabled LLMs to repair more programs and produce smaller fixes, outperforming both alternative configurations that do not employ FBFL methods and state-of-the-art symbolic program repair tools.

Formula-based fault localisation for Python. Some bugs in Python can be detected by transpiling code to C with LLMs and applying CFaults to the generated C code [49]. PyVeritas is a framework that performs high-level transpilation from Python to C, followed by bounded model checking and MaxSAT-based fault localisation through CFaults. This approach enables verification and fault localisation for Python programs using existing model-checking tools for C. Empirical evaluation on two Python benchmarks shows this method can reach accuracies of up to 80–90% for some LLMs.

9. Related Work

Fault localisation (FL) techniques typically fall into two main families: *spectrum-based (SBFL)* and *formula-based (FBFL)*. SBFL methods [50, 51, 52, 53, 54, 55] estimate the likelihood of a component being faulty based on test coverage information collected from both passing and failing test executions. While SBFL techniques are generally fast, they may lack precision, as not all components identified in this way are necessarily responsible for the observed failures [56, 57].

In contrast, FBFL approaches [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 58, 59, 60, 61, 62, 63, 64] are considered exact. As explained in Section 3, these methods (e.g., BugAssist [10], SNIPER [11]) deal with the problem of fault localisation as a *Model-Based Diagnosis (MBD)* [1] problem, seeking minimal sets of faulty components. Typically, they make a MaxSAT call for each failing test case, yielding a minimal diagnosis for each failing observation in isolation rather than reasoning over all failures jointly. They then apply an aggregation procedure to aggregate the per-test diagnoses into a single explanation of the system's inconsistency. As we show in this paper, for C software such formula-based fault localisation (FBFL) methods either produce a large number of redundant diagnoses (e.g., SNIPER) or fail to return a minimal

set sufficient to repair the program (e.g., BugAssist). Recently, Graussam [65] proposed a model-based software fault localisation approach for ANSI C, introducing a well-defined fault model that covers complex language features (including pointers and arrays) and building on CBMC encodings; consequently, the relaxation step is performed at the CBMC level rather than at the code level, as in our work. The approach also handles multiple failing observations by formulating standard MBD instances and solving them with HSD [9] to guarantee minimal diagnoses per failing test case. However, Graussam [65] applies an aggregation procedure after diagnosis enumeration that may compromise the minimality of the final diagnoses. We did not include this tool in our evaluation for two main reasons. First, in its current form, functions to be analysed must be manually annotated, an impractical requirement at our scale (we analyse over 1500 programs). Second, the tool currently supports only integers and floats in C, which makes it unsuitable for analysing the C-Pack-IPAs benchmark.

Finally, Model-Based Diagnosis has been successfully applied to restore consistency in several domains, including Boolean circuits [2, 3, 6, 7, 8, 9, 64], C software [5, 10, 11, 66], knowledge bases [12, 13, 14], and spreadsheets [15, 16]. *Program slicing* [67, 57, 68] has also emerged as a complementary technique for fault localisation in programs. A more syntactic fault localisation approach [57] employs program slicing to enumerate all minimal sets of repairs for a given faulty program. Another method for identifying the causes of faulty behaviour involves analysing the differences between successive software versions [68].

10. Conclusion

This paper introduces a novel formula-based fault localisation technique for C programs and Boolean circuits, capable of handling any number of faults. By leveraging Model-Based Diagnosis (MBD) with multiple observations, CFaults consolidates all failing test cases into a single MaxSAT formula, thereby ensuring consistency throughout the localisation process and avoiding redundant diagnoses.

Experimental evaluations highlight the effectiveness of this unified approach. On C software benchmarks, TCAS and C-Pack-IPAs, CFaults consistently outperforms existing FBFL methods such as BugAssist, SNIPER, and HSD. In particular, CFaults was both faster and able to localise more faults on C-Pack-IPAs, identifying faults in 31% more programs

than BugAssist, 36% more programs than SNIPER, and 15% more programs than HSD.

In contrast, on the ISCAS85 benchmark of Boolean circuits with multiple injected faults, CFaults is generally slower than HSD and localises faults in slightly fewer circuits overall. However, the gap is modest since CFaults localises faults in only 6% fewer circuits than HSD, demonstrating that it remains competitive in this domain.

Overall, these results establish CFaults as the fastest and most reliable FBFL approach for C software, while for Boolean circuits it remains competitive with the state-of-the-art, represented by HSD. This work thus provides the first unified framework for formula-based fault localisation across both C software and Boolean circuits domains.

Acknowledgements

PO and MK acknowledge support from the ERC under the European Union's Horizon 2020 research and innovation programme (FUN2MODEL, grant agreement No. 834115) and ELSA: European Lighthouse on Secure and Safe AI project (grant agreement No. 101070617 under UK guarantee). This work was partially supported by Portuguese national funds through FCT, under projects UID/50021/2025, UID/PRR/50021/2025, PTDC/CCI-COM/-2156/2021 (DOI: 10.54499/PTDC/CCI-COM/2156/2021), 2023.14280.PEX (DOI: 10.54499/2023.14280.PEX) and 2024.07127.CBM, and grant SFRH/-BD/07724/2020 (DOI: 10.54499/2020.07724.BD). This work was also supported by the MEYS within the program ERC CZ under the project POSTMAN no. LL1902 and co-funded by the EU under the project *ROBOPROX* (reg. no. CZ.02.01.01/00/22_008/0004590).

References

[1] R. Reiter, A theory of diagnosis from first principles, Artif. Intell. 32 (1987) 57–95. doi:10.1016/0004-3702(87)90062-2.

[2] J. de Kleer, B. C. Williams, Diagnosing multiple faults, Artif. Intell. 32 (1987) 97–130. doi:10.1016/0004-3702(87)90063-4.

[3] A. Metodi, R. Stern, M. Kalech, M. Codish, Compiling model-based diagnosis to boolean satisfaction, in: J. Hoffmann, B. Selman (Eds.),

Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada, AAAI Press, 2012, pp. 793–799. doi:10.1609/AAAI.V26I1.8222.

[4] M. Jose, R. Majumdar, Bug-assist: Assisting fault localization in ANSI-C programs, in: G. Gopalakrishnan, S. Qadeer (Eds.), Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, volume 6806 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 504–509. doi:10.1007/978-3-642-22110-1_40.

[5] P. Orvalho, M. Janota, V. Manquinho, CFaults: Model-Based Diagnosis for Fault Localization in C Programs with Multiple Test Cases, in: Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, 2024, Proceedings, volume 14933 of *Lecture Notes in Computer Science*, Springer Nature Switzerland, Cham, 2024, pp. 463–481. doi:10.1007/978-3-031-71162-6_24.

[6] R. T. Stern, M. Kalech, A. Feldman, G. M. Provan, Exploring the duality in conflict-directed model-based diagnosis, in: J. Hoffmann, B. Selman (Eds.), Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada, AAAI Press, 2012, pp. 828–834. doi:10.1609/AAAI.V26I1.8231.

[7] A. Metodi, R. Stern, M. Kalech, M. Codish, A novel sat-based approach to model based diagnosis, J. Artif. Intell. Res. 51 (2014) 377–411. doi:10.1613/JAIR.4503.

[8] J. Marques-Silva, M. Janota, A. Ignatiev, A. Morgado, Efficient model based diagnosis with maximum satisfiability, in: Q. Yang, M. J. Wooldridge (Eds.), Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015, AAAI Press, 2015, pp. 1966–1972.

[9] A. Ignatiev, A. Morgado, G. Weissenbacher, J. Marques-Silva, Model-based diagnosis with multiple observations, in: S. Kraus (Ed.), Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019, ijcai.org, 2019, pp. 1108–1115. doi:10.24963/IJCAI.2019/155.

[10] M. Jose, R. Majumdar, Cause clue clauses: error localization using maximum satisfiability, in: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, ACM, 2011, pp. 437–446.

[11] S. Lamraoui, S. Nakajima, A formula-based approach for automatic fault localization of multi-fault programs, J. Inf. Process. 24 (2016) 88–98. doi:10.2197/IPSJJIP.24.88.

[12] K. M. Shchekotykhin, G. Friedrich, P. Fleiss, P. Rodler, Interactive ontology debugging: Two query strategies for efficient fault localization, J. Web Semant. 12 (2012) 88–103. doi:10.1016/J.WEBSEM.2011.12.006.

[13] P. Rodler, K. Schekotihin, Reducing model-based diagnosis to knowledge base debugging, in: M. Zanella, I. Pill, A. Cimatti (Eds.), 28th International Workshop on Principles of Diagnosis (DX'17), Brescia, Italy, September 26-29, 2017, volume 4 of *Kalpa Publications in Computing*, EasyChair, 2017, pp. 284–296. doi:10.29007/P7ZP.

[14] P. Rodler, One step at a time: An efficient approach to query-based ontology debugging, Knowl. Based Syst. 251 (2022) 108987. doi:10.1016/J.KNOSYS.2022.108987.

[15] D. Jannach, U. Engler, Toward model-based debugging of spreadsheet programs, in: 9th Joint Conference on Knowledge-Based Software Engineering (JCKBSE 2010), 2010, pp. 252–264.

[16] P. Rodler, B. Hofer, D. Jannach, I. Nica, F. Wotawa, Choosing abstraction levels for model-based software debugging: A theoretical and empirical analysis for spreadsheet programs, Artif. Intell. 348 (2025) 104399. doi:10.1016/J.ARTINT.2025.104399.

[17] M. H. Liffiton, K. A. Sakallah, Algorithms for computing minimal unsatisfiable subsets of constraints, J. Autom. Reason. 40 (2008) 1–33. doi:10.1007/S10817-007-9084-Z.

[18] P. Rodler, Sequential model-based diagnosis by systematic search, Artif. Intell. 323 (2023) 103988. doi:10.1016/J.ARTINT.2023.103988.

[19] E. M. Clarke, D. Kroening, F. Lerda, A tool for checking ANSI-C programs, in: K. Jensen, A. Podelski (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings, volume 2988 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 168–176. doi:10.1007/978-3-540-24730-2_15.

[20] H. Do, S. G. Elbaum, G. Rothermel, Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact, Empir. Softw. Eng. 10 (2005) 405–435. doi:10.1007/S10664-005-3861-2.

[21] P. Orvalho, M. Janota, V. Manquinho, C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments, in: 2024 IEEE/ACM International Workshop on Automated Program Repair (APR), ACM, ., 2024, pp. 14–21. doi:10.1145/3643788.3648010.

[22] M. C. Hansen, H. Yalcin, J. P. Hayes, Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering, IEEE Des. Test Comput. 16 (1999) 72–80. doi:10.1109/54.785838.

[23] P. Orvalho, M. Janota, V. Manquinho, CFaults: Model-Based Diagnosis for Fault Localization in C with Multiple Test Cases, 2024. URL: https://github.com/pmorvalho/CFaults. doi:10.5281/zenodo.12510220.

[24] A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, volume 185 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2009.

[25] X. Si, X. Zhang, V. Manquinho, M. Janota, A. Ignatiev, M. Naik, On incremental core-guided maxsat solving, in: M. Rueher (Ed.), Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings, volume 9892 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 473–482. doi:10.1007/978-3-319-44953-1_30.

[26] C. M. Li, F. Manya, Maxsat, hard and soft constraints, in: Handbook of satisfiability, IOS Press, 2009, pp. 613–631.

[27] F. Bacchus, M. Järvisalo, R. Martins, Maximum Satisfiability, in: A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, IOS Press, 2021, pp. 929 – 991.

[28] A. Morgado, A. Ignatiev, J. Marques-Silva, MSCG: robust core-guided maxsat solving, J. Satisf. Boolean Model. Comput. 9 (2014) 129–134. doi:10.3233/SAT190105.

[29] A. Morgado, C. Dodaro, J. Marques-Silva, Core-guided maxsat with soft cardinality constraints, in: B. O'Sullivan (Ed.), Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings, volume 8656 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 564–573. doi:10.1007/978-3-319-10428-7_41.

[30] F. Heras, A. Morgado, J. Marques-Silva, Core-guided binary search algorithms for maximum satisfiability, in: W. Burgard, D. Roth (Eds.), Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011, AAAI Press, 2011, pp. 36–41. doi:10.1609/AAAI.V25I1.7822.

[31] J. Marques-Silva, F. Heras, M. Janota, A. Previti, A. Belov, On computing minimal correction subsets, in: F. Rossi (Ed.), IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013, IJCAI/AAAI, 2013, pp. 615–622. URL: http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6922.

[32] E. M. Clarke, D. Kroening, K. Yorav, Behavioral consistency of C and verilog programs using bounded model checking, in: Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003, ACM, 2003, pp. 368–371. doi:10.1145/775832.775928.

[33] S. Safarpour, H. Mangassarian, A. G. Veneris, M. H. Liffiton, K. A. Sakallah, Improved design debugging using maximum satisfiability, in: Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings, IEEE Computer Society, 2007, pp. 13–19. doi:10.1109/FAMCAD.2007.26.

[34] J. Davies, F. Bacchus, Solving MAXSAT by solving a sequence of simpler SAT instances, in: J. H. Lee (Ed.), Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings, volume 6876 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 225–239. doi:10.1007/978-3-642-23786-7_19.

[35] K. Chandrasekaran, R. M. Karp, E. Moreno-Centeno, S. S. Vempala, Algorithms for implicit hitting set problems, in: D. Randall (Ed.), Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011, SIAM, 2011, pp. 614–629. doi:10.1137/1.9781611973082.48.

[36] cppcheck, https://cppcheck.sourceforge.io, 2024. Accessed: 2025-09-20.

[37] clang tidy, , https://clang.llvm.org/extra/clang-tidy/, 2024. Accessed: 2025-09-20.

[38] P. Orvalho, M. Janota, V. Manquinho, GitSEED: A Git-backed Automated Assessment Tool for Software Engineering and Programming Education, in: M. Dorodchi, M. Zhang, S. Cooper (Eds.), Proceedings of the 2024 ACM Virtual Global Computing Education Conference V. 1, SIGCSE Virtual 2024, Virtual Event, NC, USA, December 5-8, 2024, ACM, 2024, pp. 165 – 171. doi:10.1145/3649165.3690106.

[39] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Trans. Program. Lang. Syst. 13 (1991) 451–490. doi:10.1145/115372.115320.

[40] A. V. Aho, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley series in computer science / World student series edition, Addison-Wesley, 1986.

[41] A. Ignatiev, A. Morgado, J. Marques-Silva, RC2: an efficient MaxSAT solver, J. Satisf. Boolean Model. Comput. 11 (2019) 53–64.

[42] A. Ignatiev, A. Morgado, J. Marques-Silva, PySAT: A python toolkit for prototyping with SAT oracles, in: O. Beyersdorff, C. M. Wintersteiger

(Eds.), Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings, volume 10929 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 428–437. doi:10.1007/978-3-319-94144-8_26.

[43] P. Orvalho, J. Piepenbrock, M. Janota, V. M. Manquinho, Graph Neural Networks for Mapping Variables Between Programs, in: ECAI 2023 - 26th European Conference on Artificial Intelligence, volume 372 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2023, pp. 1811–1818. doi:10.3233/FAIA230468.

[44] P. Orvalho, M. Janota, V. M. Manquinho, Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault Localization, in: T. Walsh, J. Shah, Z. Kolter (Eds.), AAAI-25, Sponsored by the Association for the Advancement of Artificial Intelligence, February 25 - March 4, 2025, Philadelphia, PA, USA, AAAI Press, 2025, pp. 649–657. doi:10.1609/AAAI.V39I1.32046.

[45] P. Orvalho, M. Janota, V. Manquinho, InvAASTCluster: On Applying Invariant-Based Program Clustering to Introductory Programming Assignments, J. Syst. Softw. 230 (2025) 112481. doi:10.1016/J.JSS.2025.112481.

[46] P. Orvalho, M. Janota, V. Manquinho, MENTOR: Fixing Introductory Programming Assignments with Formula-Based Fault Localization and LLM-Driven Program Repair, Journal of Systems and Software (2026). doi:https://doi.org/10.1016/j.jss.2025.112690.

[47] pycparser, , https://github.com/eliben/pycparser, 2024. [Online; accessed 2025-09-20].

[48] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, V. A. Saraswat, Combinatorial sketching for finite programs, in: J. P. Shen, M. Martonosi (Eds.), International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2006, pp. 404–415.

[49] P. Orvalho, M. Kwiatkowska, PyVeritas: On Verifying Python via LLM-Based Transpilation and Bounded Model Checking for

C, CoRR abs/2508.08171 (2025). doi:10.48550/ARXIV.2508.08171. arXiv:2508.08171.

[50] R. Abreu, P. Zoeteweij, A. J. C. van Gemund, Spectrum-based multiple fault localization, in: ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009, IEEE Computer Society, 2009, pp. 88–99. doi:10.1109/ASE.2009.25.

[51] W. E. Wong, V. Debroy, B. Choi, A family of code coverage-based heuristics for effective fault localization, J. Syst. Softw. 83 (2010) 188–208. doi:10.1016/J.JSS.2009.09.037.

[52] L. Naish, H. J. Lee, K. Ramamohanarao, A model for spectra-based software diagnosis, ACM Trans. Softw. Eng. Methodol. 20 (2011) 11:1–11:32. doi:10.1145/2000791.2000795.

[53] W. E. Wong, V. Debroy, R. Gao, Y. Li, The dstar method for effective software fault localization, IEEE Trans. Reliab. 63 (2014) 290–308. doi:10.1109/TR.2013.2285319.

[54] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A survey on software fault localization, IEEE Trans. Software Eng. 42 (2016) 707–740. doi:10.1109/TSE.2016.2521368.

[55] R. Abreu, P. Zoeteweij, R. Golsteijn, A. J. C. van Gemund, A practical evaluation of spectrum-based fault localization, J. Syst. Softw. 82 (2009) 1780–1792. doi:10.1016/J.JSS.2009.06.035.

[56] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, Y. Le Traon, You cannot fix what you cannot find! An investigation of fault localization bias in benchmarking automated program repair systems, in: 2019 12th IEEE conference on software testing, validation and verification (ICST), IEEE, 2019, pp. 102–113.

[57] B. Rothenberg, O. Grumberg, Must fault localization for program repair, in: S. K. Lahiri, C. Wang (Eds.), Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II, volume 12225 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 658–680. doi:10.1007/978-3-030-53291-8_33.

[58] J. K. Feser, S. Chaudhuri, I. Dillig, Synthesizing data structure transformations from input-output examples, in: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015, 2015, pp. 229–239.

[59] S. Lamraoui, S. Nakajima, A formula-based approach for automatic fault localization of imperative programs, in: S. Merz, J. Pang (Eds.), Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings, volume 8829 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 251–266. doi:10.1007/978-3-319-11737-9_17.

[60] A. Griesmayer, S. Staber, R. Bloem, Automated fault localization for C programs, in: R. Bloem, M. Roveri, F. Somenzi (Eds.), Proceedings of the Workshop on Verification and Debugging, V&D@FLoC 2006, Seattle, WA, USA, August 21, 2006, volume 174 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2006, pp. 95–111. doi:10.1016/J.ENTCS.2006.12.032.

[61] F. Wotawa, M. Nica, I. Moraru, Automated debugging based on a constraint model of the program and a test case, J. Log. Algebraic Methods Program. 81 (2012) 390–407. doi:10.1016/J.JLAP.2012.03.002.

[62] Y. Xie, A. Aiken, Scalable error detection using boolean satisfiability, in: J. Palsberg, M. Abadi (Eds.), Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005, ACM, 2005, pp. 351–363. doi:10.1145/1040305.1040334.

[63] R. Könighofer, R. Bloem, Automated error localization and correction for imperative programs, in: P. Bjesse, A. Slobodová (Eds.), International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011, FMCAD Inc., 2011, pp. 91–100.

[64] H. Zhou, D. Ouyang, X. Zhao, L. Zhang, Two compacted models for efficient model-based diagnosis, in: Thirty-Sixth AAAI Conference on

Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022, AAAI Press, 2022, pp. 3885–3893. doi:10.1609/AAAI.V36I4.20304.

[65] L. Graussam, Consistency-based Software Fault Localization with Multiple Observations, Diplom-Ingenieur Thesis, Technische Universität Wien, 2024.

[66] P. M. O. M. da Silva, MENTOR: Automated Feedback for Introductory Programming Exercises, Ph.D. thesis, INSTITUTO SUPERIOR TÉCNICO, 2025.

[67] E. O. Soremekun, L. Kirschner, M. Böhme, A. Zeller, Locating faults with program slicing: an empirical analysis, Empir. Softw. Eng. 26 (2021) 51. doi:10.1007/S10664-020-09931-7.

[68] A. Zeller, Yesterday, my program worked. Today, it does not. Why?, in: O. Nierstrasz, M. Lemoine (Eds.), Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Proceedings, volume 1687 of *Lecture Notes in Computer Science*, Springer, 1999, pp. 253–267.

Appendix A. Experiments on C Software Without Hierarchical Weights

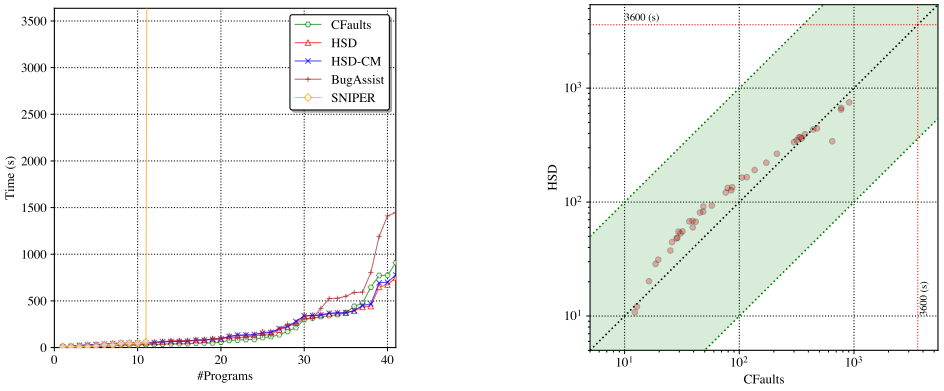
In this section, we evaluate all FBFL approaches on both C benchmarks, TCAS [20] and C-Pack-IPAs [21], *without* hierarchical weights on the soft clauses (relaxation variables; see Section 5.1.4). Results *with* hierarchical weights are reported in Section 6.1.

Appendix A.1. TCAS

Table A.6 summarises the results for BugAssist, SNIPER, HSD, HSD-CM, and CFaults on TCAS *without* hierarchical weights. Entries highlighted in bold correspond to the highest value per column. The outcomes are similar to those obtained *with* hierarchical weights (see Table 3), all FBFL methods, except SNIPER, find valid diagnoses for every instance in TCAS.

Benchmark: **TCAS**			
	Valid Diagnosis	**Memouts**	**Timeouts**
BugAssist	**41 (100.0%)**	0 (0.0%)	0 (0.0%)
SNIPER	11 (26.83%)	**30 (73.17%)**	0 (0.0%)
HSD	**41 (100.0%)**	0 (0.0%)	0 (0.0%)
HSD-CM	**41 (100.0%)**	0 (0.0%)	0 (0.0%)
CFaults	**41 (100.0%)**	0 (0.0%)	0 (0.0%)

Table A.6: BugAssist, SNIPER, HSD and CFaults fault localisation results on TCAS [20] benchmark, not using hierarchical weights.



(a) Time performance of all FBFL approaches.

(b) Time comparison between CFaults and HSD.

Figure A.7: Total CPU time comparison between BugAssist, SNIPER, HSD and CFaults on the TCAS [20] benchmark, not using hierarchical weights.

Benchmark: **C-Pack-IPAs**			
	Valid Diagnosis	**Memouts**	**Timeouts**
BugAssist	284 (41.64%)	4 (0.59%)	394 (57.77%)
SNIPER	248 (36.36%)	**21 (3.08%)**	**413 (60.56%)**
HSD	382 (56.01%)	11 (1.61%)	289 (42.38%)
HSD-CM	369 (54.11%)	12 (1.76%)	301 (44.13%)
CFaults	**469 (68.77%)**	15 (2.20%)	198 (29.03%)

Table A.7: BugAssist, SNIPER, HSD and CFaults fault localisation results on C-Pack-IPAs [21] benchmark, without hierarchical weights.

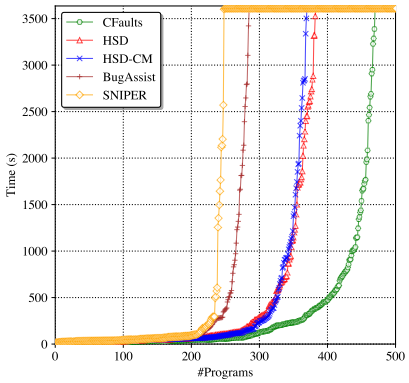
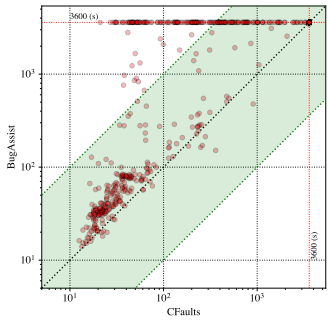


Figure A.8: Total CPU time performance of all FBFL approaches on C-Pack-IPAs, not using hierarchical weights.

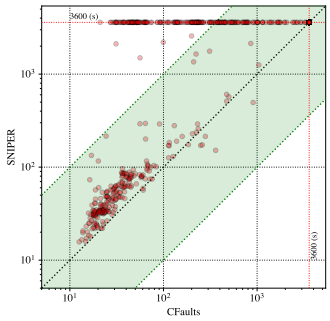
Figure A.7a shows a *cactus plot* of CPU time (y-axis) versus the number of successfully localised faulty programs (x-axis) for all FBFL algorithms on TCAS without hierarchical weights. CFaults follows the same trend as in the hierarchical weighted setting: it is faster than BugAssist and SNIPER, and typically slightly faster than HSD. Figure A.7b presents a *scatter plot* comparing CFaults and HSD CPU times on TCAS without hierarchical weights. Each point corresponds to a faulty program; the x-coordinate (resp. y-coordinate) is the time taken by CFaults (resp. HSD) to localise the fault. Points above the diagonal indicate cases where HSD required more time than CFaults. As shown, CFaults is generally faster than HSD on TCAS even in the unweighted setting.

Appendix A.2. C-Pack-IPAs

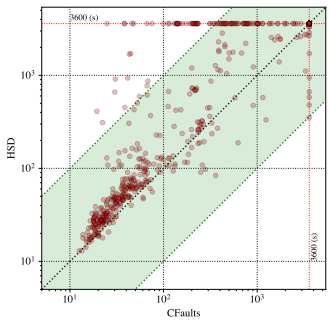
Table A.7 summarises results for BugAssist, SNIPER, HSD, HSD -CM, and CFaults on C-Pack-IPAs *without* hierarchical weights. Entries highlighted in bold correspond to the highest value per column. Figure A.8 provides a cactus plot of total CPU time across all FBFL algorithms. Without hierarchical weights, CFaults localises faults in 28% more programs than BugAssist, 33% more than SNIPER, and 13% more than HSD (using HSD's best configuration on C-Pack-IPAs, i.e., without core minimisation). These outcomes are similar to those obtained *with* hierarchical weights (see Table 3). Figure A.9 presents scatter plots comparing CFaults with each competing method in terms of total CPU time, further showing that, in general, CFaults is faster than the other FBFL approaches when localising faults in the same C programs without hierarchical weights.



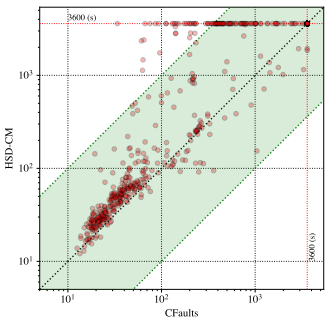
(a) CFaults vs BugAssist.



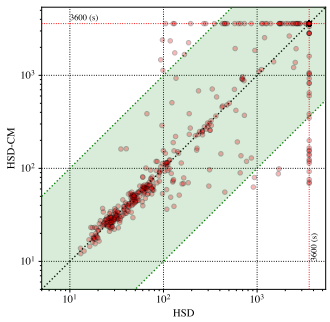
(b) CFaults vs SNIPER.



(c) CFaults vs HSD.



(d) CFaults vs HSD-CM.



(e) HSD vs HSD-CM.

Figure A.9: CPU time comparison between BugAssist, SNIPER, HSD, HSD with core minimisation (HSD-CM) and CFaults on C-Pack-IPAs, not using hierarchical weights.