Quantitative Robustness for Vulnerability Assessment

GUILLAUME GIROL, Université Paris-Saclay, CEA, List, France
GUILHEM LACOMBE, Université Paris-Saclay, CEA, List, France
SÉBASTIEN BARDIN, Université Paris-Saclay, CEA, List, France

Most software analysis techniques focus on bug reachability. However, this approach is not ideal for security evaluation as it does not take into account the difficulty of triggering said bugs. The recently introduced notion of *robust reachability* tackles this issue by distinguishing between bugs that can be reached independently from uncontrolled inputs, from those that cannot. Yet, this *qualitative* notion is too strong in practice as it cannot distinguish mostly replicable bugs from truly unrealistic ones. In this work we propose a more flexible *quantitative* version of robust reachability together with a dedicated form of symbolic execution, in order to automatically measure the difficulty of triggering bugs. This *Quantitative Robust Symbolic Execution (QRSE)* relies on a variant of model counting, which allows to account for the asymmetry between attacker-controlled and uncontrolled variables. While this specific model counting problem has been studied in AI research fields such as Bayesian networks, knowledge representation and probabilistic planning, its use within the context of formal verification presents new challenges. We show the applicability of our solutions through security-oriented case studies, including real-world vulnerabilities such as CVE-2019-20839 from libvncserver.

CCS Concepts: • **Security and privacy** → **Formal methods and theory of security**; **Logic and verification**; *Software security engineering*; • **Theory of computation** → *Program reasoning*; **Program analysis**.

Additional Key Words and Phrases: automated verification, static analysis, symbolic execution

1 INTRODUCTION

Context & Problem. Many software analysis problems are reduced to the *reachability* of a specific condition, for example local assertions indicating bugs such as buffer overflows or null pointer dereference. Yet this notion of reachability is too weak for the purpose of vulnerability assessment as it only proves that the bug exists within a particular context but does not ensure that the latter can be *reliably* triggered by an attacker. For example, a bug may require a specific stack base address, while it is out of the attacker's control when protections such as ASLR[1] are enabled.

Recent works [Girol et al., 2021, 2022] introduced the stronger notion of *robust reachability* to determine whether attackers can reproduce a bug reliably. A bug is *robustly reachable* if attackers can choose some program inputs (under their control) that triggers the bug with absolute certainty regardless of the value of the other input (outside their control). Unfortunately, robust reachability may be too strong in practice, as it can only highlight bugs for which optimal attackers will obtain systematic success, while a high-probability success is often sufficient. There is thus a need for a more nuanced notion of robustness, together with appropriate tooling.

Goal & Challenges. Our work aims to enable quantitative assessment of the success rate of an attack by developing a *quantitative* counterpart to robust reachability, similarly to quantitative information flow [Heusser and Malacaria, 2010] for non-interference [Goguen and Meseguer, 1982] or to the shift from model checking to probabilistic model checking [Aziz et al., 1996].

Proposal. We split program inputs into attacker-controlled inputs a and uncontrolled inputs x. We define *quantitative robustness* as the maximum proportion of uncontrolled inputs $x \in X$ triggering the bug with a fixed controlled input $a \in A$.

[1] Address Space Layout Randomization.

Authors' addresses: Guillaume Girol, Université Paris-Saclay, CEA, List, Saclay, France, guillaume.girol@cea.fr; Guilhem Lacombe, Université Paris-Saclay, CEA, List, Saclay, France, guilhem.lacombe@cea.fr; Sébastien Bardin, Université Paris-Saclay, CEA, List, Saclay, France, sebastien.bardin@cea.fr.

In this work, we set up the stage for an automated formal treatment of quantitative robustness, providing formal definition, studying its properties and proposing a bounded-verification algorithm based on symbolic execution. This algorithm relies on the ability to compute path-wise quantitative robustness, which is *both* a counting and an optimization problem, as we need to find the maximum number of uncontrolled inputs triggering the bug for a fixed controlled input. This underlying counting problem is thus very different from those commonly used in quantitative verification, such as model counting [Valiant, 1979] and projected model counting [Aziz et al., 2015]. Still, it has already been studied for the propositional case in some AI-related fields under the name of f**-E-MAJSAT** [Littman et al., 1998].

Unfortunately, existing solvers [Fremont et al., 2017, Huang, 2006, Lee et al., 2018, Majercik and Boots, 2005, Pipatsrisawat and Darwiche, 2009] are optimized for specific application domains (e.g., probabilistic planning), which is often detrimental for our purpose. We thus propose a new parametric approximate algorithm specially optimized for quantitative robustness.

Contributions. We claim the following contributions:

- We define a quantitative pendant of robust reachability called quantitative robustness (Section 4), which generalize both reachability and robust reachability. Interestingly, quantitative robustness behaves better on branches than robust reachability, allowing incremental reasoning. We also discuss the relationship with existing quantitative formalisms such as probabilistic temporal logics and games;
- We propose *Quantitative Robust Symbolic Execution (QRSE)* (Section 5), a variant of symbolic execution for computing quantitative robustness. We show that *QRSE* does not strictly require path merging, contrary to robust reachability analysis, re-establishing the traditional symmetry in deduction power between symbolic execution [Cadar and Sen, 2013] and bounded model checking [Clarke et al., 2004] which was broken in the case of robust reachability;
- We propose a method reducing path-wise quantitative robustness with finite variable domains (e.g., bitvectors and arrays) to f**-E-MAJSAT** (Section 6), a counting problem studied in some sub-fields of AI. As off-the-shelf methods turn out to be too inefficient or imprecise for our purposes, we introduce a novel parametric algorithm allowing to tune the trade-off between precision and performance via a novel (approximated) technique we call *relaxation* (Section 7.3);
- We have implemented these ideas in two tools: BINSEC/QRSE and the Popcon solver (Section 8). First experiments on security-oriented case studies (including real-world vulnerabilities such as CVE-2019-20839 from libvncserver) demonstrate that *QRSE* enables fine-grained bug triage compared with symbolic execution and robust symbolic execution, and that relaxation solves more problems arising from *QRSE* than prior techniques while minimizing approximation.

Quantitative robustness is a new approach to assess the replicability of a bug, offering better flexibility than its qualitative counter part. We believe this is an interesting step toward security-relevant *quantitative* program analysis. Surprisingly, while quantitative robustness possibly opens new opportunities for formal methods in security analysis, it also draws new connections with notions originating from different fields of AI.

2 MOTIVATING EXAMPLE

Consider Figure 1, a case of two network servers incorrectly using uninitialized memory to determine the privileges of clients, loosely inspired by a vulnerability in the `doas` sudo-like

command (CVE-2019-15900). Whether a client can perform a sensitive operation depends on a `privilege_level` which is accessed through the `get_privilege_level` getter. We want to analyze a bug causing this getter to incorrectly return uninitialized memory.

```
/* main privilege levels */
#define DEFAULT_LEVEL 1
#define OPERATOR_LEVEL 100
#define ADMIN_LEVEL 9000
/* commands */
#define DROP_PRIVILEGE 0
#define DROP_PRIVILEGE_LEGACY 1
#define GET_VERSION 2
#define SUDO 3

uint32_t uninit; /* random data */
uint32_t privilege_level = DEFAULT_LEVEL;

void set_privilege_level(uint32_t new) {
  privilege_level = new;
}

uint32_t get_privilege_level() {
  // bug: return uninitialized memory
  return uninit;
}
```

```
void prog1(uint32_t command, uint32_t argument) {
  if (command == GET_VERSION) {
    /* harmless */
  } else {
    /* command is sudo */
    if (get_privilege_level() == OPERATOR_LEVEL) {
      set_privilege_level(ADMIN_LEVEL);
    }
  }
}

void prog2(uint32_t command, uint32_t argument) {
  switch (command) {
    case GET_VERSION: /* harmless */ break;
    case DROP_PRIVILEGE:
    case DROP_PRIVILEGE_LEGACY:
      if (argument<get_privilege_level()) {
        set_privilege_level(argument);
      }
  }
}
```

Fig. 1. `prog1` and `prog2` are both vulnerable, but the second more than the first

We consider network attackers who can send one request under the form of a pair (`command`, `argument`) passed to either `prog1` or `prog2` depending on the version of the server. They cannot influence other parameters, most notably uninitialized memory `uninit`.

Can attackers obtain an equal or greater privilege level than `ADMIN_LEVEL` *by submitting a carefully chosen command and argument?*

prog1: happens when the formula $f_1 \triangleq \texttt{command} \neq 2 \wedge \texttt{uninit} = 100$ is satisfied.

prog2: same with $f_2 \triangleq \texttt{command} \in \{0, 1\} \wedge 9000 \leq \texttt{argument} < \texttt{uninit}$.

In `prog1`, when attackers play perfectly by choosing $\texttt{command} = 1$, they need to be lucky: only one value of `uninit` out of 2^{32} lets them win. However in `prog2`, for $\texttt{command} = 1$ and $\texttt{argument} = 9000$ more than 99% of the values that `uninit` can take will let attackers achieve their goal. Our goal is to develop a method which allows to automatically recognize this fact.

Qualitative Methods. Traditional bug finding techniques based on standard reachability, such as symbolic execution [Schwartz et al., 2010] or bounded model checking [Biere et al., 2003], are of little use here, as both attacks are *reachable, i.e.* f_1 and f_2 both admit at least one solution. On the other hand, robust reachability [Girol et al., 2021, 2022] requires that the attack always works for at least one controlled input: $\exists \texttt{command}, \texttt{argument}. \forall \texttt{uninit}. f_x$. In this case it is too strict as neither `prog1` nor `prog2` satisfy it. Overall, qualitative methods based on either standard reachability or robust reachability cannot distinguish between the two attacks.

Standard Model Counting. Where these *qualitative* techniques fail to distinguish both vulnerabilities, a more *quantitative* one may bear fruit. For example, we can compare the number of solutions of f_1 and f_2, or rather their density in a search space of size 2^{96}. This is reminiscent of probabilistic symbolic execution [Geldenhuys et al., 2012]. For f_1, this density is $\frac{(2^{32}-1)\times 2^{32}}{2^{96}} \simeq 2.3 \cdot 10^{-10}$, and for

f_2 it is $\frac{(2^{32}-9001)(2^{32}-9000)}{2^{96}} \simeq 2.3 \cdot 10^{-10}$. Unfortunately, these values are very close, and worse, they compare in order opposite to what we expect: $f_1 > f_2$.

Our Approach. The missing ingredient here is to take into account the threat model: in the worst case attackers will choose the best possible input, *i.e.* `command = 1` and `argument = 9000`, but they cannot influence the value of `uninit`. We need to compute the number of solutions for the value of `command` and `argument` most favorable to attackers:

$$\textbf{prog1} \qquad \max_{\substack{\texttt{command}\\ \texttt{argument}}} |\{\texttt{uninit} \mid f_1\}| = |\{100\}| = 1 \tag{1}$$

$$\textbf{prog2} \qquad \max_{\substack{\texttt{command}\\ \texttt{argument}}} |\{\texttt{uninit} \mid f_2\}| = |[9001; 2^{32}-1]| = 2^{32} - 9001 \tag{2}$$

These numbers can be compared fairly as the search space has the same size (2^{32}). In the general case we will consider a proportion of inputs, which we call *quantitative robustness*. Quantitative robustness does align with our intuition: it is low ($2.3 \cdot 10^{-10}$) for `prog1` but high (~ 0.9999979043) for `prog2`, showing that the attack on `prog2` is much more replicable that the one on `prog1`.

The problem on boolean formulas in eqs. (1) and (2) is known as f**-E-MAJSAT** [Littman et al., 1998]. A few dedicated solvers have been developed in the AI community but, although some of them [Huang, 2006, Majercik and Boots, 2005] can obtain eq. (1) in a few seconds, none we have tried could obtain eq. (2) even at the price of reasonable approximation.

Taking inspiration from existing knowledge-compilation based algorithms, we propose a new technique called relaxation that offers an interesting trade-off between performance and precision. Coupling this solver with a quantitative variant of symbolic execution, we are able to automatically estimate the quantitative robustness of the two programs. For `prog2` we obtain that the quantitative robustness of privilege escalation is comprised between 0.9963 and 1 in about 1 second. This is enough to conclude that there are many more initial states that let attackers exploit the vulnerability in `prog2` than in `prog1`. We interpret this as a sign that this bug is likely more severe in `prog2` than in `prog1`.

Conclusion. Qualitative program analysis techniques based on reachability and robust reachability cannot distinguish `prog1` from `prog2`, whereas in practice attackers have many more opportunities to trigger the bug in `prog2`. Quantitative robustness clearly discriminates between the two, but this is not only because it is quantitative. Compared to probabilistic symbolic execution [Geldenhuys et al., 2012], quantitative robustness fits security contexts better by using a variant of model counting which can distinguish between attacker-controlled inputs and uncontrolled inputs.

3 BACKGROUND

A program P is represented as a transition system with transition relation $\rightarrow$ over the set of states $\mathcal{S}$. A trace is a succession of states respecting $\rightarrow$; the set of traces of a program P is $T(P)$. Each state has a corresponding location in the code of the program, a *path* is a succession of locations. The initial state is determined by the input y of the program and $P|_y$ is a program identical to P but executed on input y.

Reachability. For O a set of finite traces, we say that O is *reachable* in P when $T(P) \cap O \neq \varnothing$, meaning that P admits a trace reaching the goal.

Robust Reachability. Let input y be a pair (a, x) of *controlled inputs* $a \in \mathcal{A}$ chosen by attackers and *uncontrolled inputs* $x \in \mathcal{X}$ unknown to attackers and uninfluenced by them. We say that O is

robustly reachable [Girol et al., 2021, 2022] when $\exists a \in \mathcal{A}. \forall x \in \mathcal{X}. T\left(P|_{(a,x)}\right) \cap O \neq \varnothing$, meaning that for some controlled input a the target is reached regardless of inputs x.

```
   Data: bound k, target O
1  for path π in GetPaths (k) do
2  |  φ := GetPredicate(π, O)
3  |  if ∃i. φ(i) then return true
4  end
5  return false
```

Algorithm 1: Reachability of O by symbolic execution

Symbolic Execution. Reachability can be proved by *Symbolic Execution (SE)* [Cadar and Sen, 2013], as shown on Algorithm 1. *SE* enumerates all paths π and converts them to SMT formulas[2] $\mathrm{pc}_\pi^O(i)$ called *path constraints* expressing the constraints over inputs i triggering π and reaching the goal O. It then checks whether this formula is satisfiable, in which case O is reachable. *SE* is *correct*, i.e. detected targets are reachable, and *k-complete*, i.e. any reaching path of length up to k is detected.

Robust Symbolic Execution. *Robust Symbolic Execution (RSE)* [Farinier et al., 2018a, Girol et al., 2021, 2022] proves robust reachability by replacing satisfiability tests $\exists a, x.\ \mathrm{pc}_\pi^O(a, x)$ in *SE* by $\exists a. \forall x.\ \mathrm{pc}_\pi^O(a, x)$, i.e. robust reachability. It is correct, but not k-complete. For k-completeness, path merging [Hansen et al., 2009] is required: path constraints are merged together as $\bigvee_i \mathrm{pc}_{\pi_i}^O(a, x)$.

4 QUANTITATIVE ROBUSTNESS

In this section, we provide a generic framework for quantitative robustness, with potentially infinite input domains and including various useful properties.

4.1 Threat Model

We adopt same threat model as Girol et al. [2021, 2022], with inputs split between *controlled inputs* $a \in \mathcal{A}$ and *uncontrolled inputs* $x \in \mathcal{X}$. The goal of attackers is to reach a finite set of traces corresponding to a vulnerability. We assume that programs are deterministic, although sources of randomness can be modeled as uncontrolled inputs.

This threat model fits the scenario of attackers sending a single request to a non-interactive system in order to perform some sort of exploit. In particular, the underlying vulnerability may not be reliably triggerable, requiring multiple attempts depending on the chosen input. For example, attackers may attempt to exploit an uninitialized variable, such as in our motivating example from Figure 1, by repeatedly sending requests until its value is exploitable. Since smart attackers try to maximize their chances, analysis must focus on the worst case scenario, i.e. an optimal input.

The downside of this threat model is that it excludes interactive systems, however it ensures proof methods remain tractable.

4.2 Formal Definition

We define quantitative robustness as the maximal proportion of uncontrolled inputs that reaches the target for a fixed controlled input. In order to handle infinite input domains, i.e. natural n-uples, we thus need to define a notion of density of a subset. One way of achieving this is to extend the notion of *asymptotic density* to n-uples.

[2]Typically, quantifier-free conjunctive formulas over some theories such as integers, bitvectors, arrays, etc.

Definition 4.1 (Asymptotic Density for n-uples). Let $Y \subset X \subset \mathbb{N}^n$. When $n = 1$, the asymptotic density of Y in X is $d_X(Y) = \lim_{i\to\infty} \frac{|\{y \in Y / y<i\}|}{|\{x \in X / x<i\}|}$. We extend this definition for $n > 1$ as the following:

$$d_X(Y) \triangleq \lim_{i\to\infty} \frac{|\{(y_1, ..., y_n) \in Y / y_1 + ... + y_n < i\}|}{|\{(x_1, ..., x_n) \in \mathcal{X} / x_1 + ... + x_n < i\}|}$$

We can now give our formal definition of quantitative robustness. Let us consider a program P and a target set of finite traces O.

Definition 4.2 (Quantitative robustness). We define the quantitative robustness $q(P, O)$ of O in P as the following:

$$q(P, O) \triangleq \sup_{a\in\mathcal{A}} d_{\mathcal{X}}\left(\left\{x \in \mathcal{X} \mid T\left(P|_{(a,x)}\right) \cap O \neq \varnothing\right\}\right)$$

Proposition 4.3. *If both $\mathcal{X}$ and $\mathcal{A}$ are finite, we get the following:*

$$q(P, O) \triangleq \frac{1}{|\mathcal{X}|} \max_{a\in\mathcal{A}} \left|\left\{x \in \mathcal{X} \mid T\left(P|_{(a,x)}\right) \cap O \neq \varnothing\right\}\right|$$

In the case of our motivating example (Figure 1), we thus get $q\left(prog1, O_{prog1}\right) = \frac{1}{2^{32}}$ and $q\left(prog2, O_{prog2}\right) = \frac{2^{32}-9001}{2^{32}}$.

Proposition 4.4. *Extreme values of quantitative robustness correspond to already known properties:*

$$q(P, O) = 0 \iff O \text{ is not reachable}$$
$$q(P, O) = 1 \iff O \text{ is robustly reachable}$$

Intuitively, quantitative robustness allows to distinguish bugs which are mostly robust from those which are not, i.e. quantitative robustness is measured above a *tolerance threshold* given by the user – typically a security expert. For example, in our case study from Section 8.5 we consider a quantitative robustness of 0.2 highly concerning (i.e., in practice, deserving further attention by a security expert), and a quantitative robustness lower than 10^{-6} as mere noise.

Scope & Limitations. This definition inherits some limitations of robust reachability. As mentioned previously, interactive systems are not accounted for. In addition, we limit our discussion to the reachability of a (possibly infinite) set of finite traces, which already encompasses critical scenarios such as buffer and stack overflows, use-after-free and control-flow hijacking. Handling more advanced properties such as hyperproperties (e.g., secret leakages) or infinite traces (e.g., denial of service) is left for future work, although it should be straightforward for hyper-safety properties such as non-interference.

In addition, this approach assumes that uncontrolled inputs are uniformly distributed, which may not be the case in practice. However this is a common assumption in quantitative analysis, even though it is not always explicitly stated. Also, determining input distributions is certainly a challenge of its own in practice. Tackling these issues is out of the scope of this work.

4.3 Quantitative Robustness and Path Merging

One limitation of robust reachability is the need for path merging [Girol et al., 2021], which complicates program analysis in practice. This issue can be mitigated for quantitative robustness thanks to some properties we will now highlight.

Robust reachability can be lost at a branch depending on uncontrolled input and recovered later when paths meet again. Consider Figure 2, a case where path merging is necessary for robust reachability. The program P has two paths π_1 and π_2 starting at a location s selected depending on an uncontrolled boolean input x and joining again at location ℓ. Neither π_1 nor π_2 satisfy robust

```
void main(a, x) {
  if (x) x++; // π1
  else x--;   // π2
  if (!a) bug(); // ℓ
}
```

Fig. 2. An example where path merging is required in *RSE* (taken from Girol et al. [2021])

reachability, yet ℓ is robustly reachable. Robust reachability can thus "reappear" from non-robust paths quite unpredictably, thus we are forced to merge all paths for the sake of completeness.

However *this is not the case with quantitative reachability* due to the following property:

Proposition 4.5 (Quantitative robustness pseudo-conservation). *Let $\pi_1, \ldots, \pi_n$ be paths in a program P and $P|^{\pi_1,\ldots,\pi_n}$ the restriction of P to $\pi_1, \ldots, \pi_n$. There exists $1 \leq i \leq n$ such that:*

$$q\left(P|^{\pi_i}, O\right) \geq \frac{1}{n} q\left(P|^{\pi_1,\ldots,\pi_n}, O\right)$$

Proof. For convenience, we note:
$\mathit{Reaching}(P, a, O) \triangleq \left\{x \in \mathcal{X} \mid T\left(P|_{(a,x)}\right) \cap O \neq \varnothing\right\}$ and $R_\pi(a) \triangleq \mathit{Reaching}\left(P|^{\pi}, a, O\right)$.

Lemma 4.6 (Quantitative robustness of merged paths). *Let π and π' be two paths in a program P.*

$$q\left(P|^{\pi,\pi'}, O\right) \leq q\left(P|^{\pi}, O\right) + q\left(P|^{\pi'}, O\right)$$

Proof of the lemma. For any $a \in \mathcal{A}$, $R_{\pi,\pi'}(a) = R_\pi(a) \cup R_{\pi'}(a)$ by def. and $d_{\mathcal{X}}(R_{\pi,\pi'}(a)) \leq d_{\mathcal{X}}(R_\pi(a)) + d_{\mathcal{X}}(R_{\pi'}(a))$. Thus $\sup_{a\in\mathcal{A}} d_{\mathcal{X}}(R_{\pi,\pi'}(a)) \leq \sup_{a\in\mathcal{A}} d_{\mathcal{X}}(R_\pi(a)) + \sup_{a\in\mathcal{A}} d_{\mathcal{X}}(R_{\pi'}(a))$ and $q\left(P|^{\pi,\pi'}, O\right) \leq q\left(P|^{\pi}, O\right) + q\left(P|^{\pi'}, O\right)$. □

By contradiction, if $q\left(P|^{\pi_i}, O\right) < \frac{1}{n} q\left(P|^{\pi_1,\ldots,\pi_n}, O\right)$ for all i from 1 to n, then by Lemma 4.6, $q\left(P|^{\pi_1,\ldots,\pi_n}, O\right) < n \times \frac{1}{n} q\left(P|^{\pi_1,\ldots,\pi_n}, O\right)$ which is absurd. □

In our example, either π_1 or π_2 has quantitative reachability at least $\frac{1}{2}$, thus ℓ can still be detected without path merging by halving the detection threshold. We will take advantage of this property to avoid resorting to path merging in Section 5.

5 QUANTITATIVE ROBUST SYMBOLIC EXECUTION

In this section, we propose a method to enumerate all locations with quantitative robustness above a given threshold Q. Our solution is based on symbolic execution, with an oracle `ComputePQR` (P, π, O) able to compute the Path-wise Quantitative Robustness $q\left(P|^{\pi}, O\right)$ of any path π. While we consider here possibly infinite domains, we show how to compute `ComputePQR` (P, π, O) for finite input sets in Sections 6 and 7.

5.1 From *RSE* to *QRSE*

We adapt *RSE* [Girol et al., 2021, 2022] by replacing the universal satisfiability test $\exists a. \forall x.\ \mathrm{pc}^O_\pi(a, x)$ with the quantitative robustness test `ComputePQR`$(P, \pi, O) \geq Q$. This way we can enumerate paths reaching the goal with quantitative robustness above the threshold Q. We call this technique *QRSE*. More specifically, operating this substitution on *RSE* yields *QRSE* (Algorithm 2) and on *RSE with path merging (RSE+)* yields *QRSE with path merging (QRSE+)* (Algorithm 3).

Data: bound k, target O, threshold Q
for *path* π *in* `GetPaths` (k) **do**
  $\chi :=$ `ComputePQR`(P, π, O)
  if $\chi \geq Q$ **then**
    /* O has quantitative robustness $\geq \chi$ */
    return (true, χ)
end
return false

Algorithm 2: *QRSE*

Data: bound k, target O, threshold Q
$paths = \varnothing$
for *path* π *in* `GetPaths` (k) **do**
  $paths := paths \cup \{\pi\}$
  $\chi :=$ `ComputePQR`$(P, paths, O)$
  if $\chi \geq Q$ **then**
    /* O has quantitative robustness $\geq \chi$ */
    return (true, χ)
end
return false

Algorithm 3: QRSE+ (with path merging)

Proposition 5.1 (Correctness of *QRSE* and *QRSE+*). *If QRSE or QRSE+ reports a target O with quantitative robustness χ, then $q(P, O) \geq \chi$.*

Proof.

Lemma 5.2 (Monotonicity of quantitative robustness of paths). *Let π be a path in a program P.*

$$q(P|^{\pi}, O) \leq q(P, O)$$

Proof of the lemma. *Reaching*$(P|^{\pi}, a, O) \subseteq$ *Reaching*$(P, a, O)\ \forall a \in \mathcal{A}$, thus $q(P|^{\pi}, O) \leq q(P, O)$ by definition. □

If *QRSE* reaches O, there is a path π s.t. $q(P|^{\pi}, O) = \chi$. Then by Lemma 5.2, $q(P, O) \geq \chi$. □

Proposition 5.3 (k-completeness of *QRSE+*). *We note $P|^{\leq k}$ the restriction of program P to traces of length at most k. Let Q be a detection threshold. Assuming solver termination, if $q\left(P|^{\leq k}, O\right) \geq Q$ then QRSE+ reports O with quantitative robustness χ between Q and $q\left(P|^{\leq k}, O\right)$.*

Proof. In $P|^{\leq k}$, for each possible input, there is at most one maximal path of length at most k (and all its prefixes). When *QRSE+* has explored all paths, the path constraint will be equivalent to reaching O. The oracle on the merged path constraint will therefore return the desired value $q\left(P|^{\leq k}, O\right)$. If a subset of paths has quantitative robustness between Q and $q\left(P|^{\leq k}, O\right)$, *QRSE+* may return early. □

Approximations. If we can only approximate $q(P|^{\pi}, O)$ with `ComputePQR`(P, π, O), we retain some guarantees: with a lower bound *QRSE* is still correct and with an upper bound *QRSE+* is still k-complete.

5.2 Getting Rid of Path Merging

RSE requires path merging for k-completeness [Girol et al., 2021]. In addition, Proposition 5.3 suggests that it may also be the case for *QRSE*. However, we want to avoid it for two main reasons: first, some paths can be hard to execute symbolically (*e.g.* because they contain exotic system calls, dynamic jumps, *etc.*), and second, merged path constraints are often more complex and harder to solve. In the quantitative case, we can show that *QRSE* (no path merging) is as complete as *QRSE+* (path merging) *under a reasonable assumption* (Assumption 5.4).

Assumption 5.4 (Badly scaling path merging assumption). *We assume that merged paths constraints are more difficult to solve than their constituents [Hansen et al., 2009, Kuznetsov et al., 2012], and that there is an integer κ such that, when merging the paths constraints of more than κ paths together, the resulting path constraint is so large and/or complex that our implementation of the oracle* `ComputePQR` *will return UNKNOWN.*

Proposition 5.5 (*QRSE* vs *QRSE+*). *Under the badly scaling path merging assumption, all locations reported by QRSE+ as having quantitative robustness above the threshold Q are also reported by QRSE with the threshold Q/κ.*

Proof. Let O be a target reported by *QRSE+* with threshold Q. By the badly scaling path merging assumption (Assumption 5.4), there are paths $\pi_1, \ldots, \pi_n$ with $n \leq \kappa$ *s.t.* the oracle can compute $\chi \triangleq \texttt{ComputePQR}(P, \pi_1, \ldots, \pi_n, O)$ with $\chi \geq Q$. By Proposition 4.5, there is a path π_i such that $q\,(P|^{\pi_i}, O) \geq Q/n \geq Q/\kappa$. Thus *QRSE* detects O through path π_i with the threshold Q/κ. □

In practice, this means that if path merging turns out to be a problem for *QRSE+* with threshold Q, then one can run *QRSE* with threshold Q/κ and have the guarantee of finding all targets with quantitative robustness above Q but no targets with quantitative robustness below Q/κ. The second point ensures we keep a good signal-to-noise ratio. This principle will be illustrated in our second case study about `libvncserver` (Section 8.6).

6 PATH-WISE QUANTITATIVE ROBUSTNESS AS A COUNTING PROBLEM

We now focus on a practical way of implementing the `ComputePQR` (P, π, O) for Path-wise Quantitative Robustness. *We will restrict ourselves from now on to the case of finite input domains.* We present the technique for propositional logic, yet the technique itself can be applied to other theories that reduces to it, for example bitvectors with arrays and floats.

The main result here is to show that computing quantitative robustness for a single path reduces in this setting to a variant of model counting called f**-E-MAJSAT**. We will then provide in Section 7 a novel approximation technique for f**-E-MAJSAT**.

6.1 Preliminary: Propositional Formulas

The set $\mathcal{F}$ of propositional formulas is defined recursively as variables $v \in \mathcal{V}$ and negations $(\neg f)$, conjunctions $(f \wedge g)$ and disjunctions $(f \vee g)$ of formulas $f, g \in \mathcal{F}$. We note $V(f)$ the set of variables appearing in a formula f.

A *literal* is a variable or the negation of a variable and a *clause* is a finite conjunction or disjunction of literals. Propositional formulas are usually given in *Conjunctive Normal Form (CNF), i.e.* conjunctions of disjunctive clauses.

A *partial valuation* is a partial mapping from a subset of $\mathcal{V}$ to the set of booleans $\mathbb{B} \triangleq \{\top, \bot\}$. We note $f|_m$ the application of a partial valuation m to a formula f, where variables v in the domain of m are replaced by $m(v)$. For example, for $f = v_1 \wedge (\neg v_1 \vee v_2)$ and $m = \{v_1 \mapsto \top\}$ we get $f|_m = v_2$. A valuation is *complete* for f when its domain contains $V(f)$, *i.e.* it associates all variables to a boolean value; it thus maps f itself to a boolean.

A complete valuation m is a model of a formula f if $f|_m = \top$. We note $M(f) \triangleq \{m \in \mathbb{B}^{V(f)} \mid f|_m = \top\}$ the set of models of a formula f and $\sharp(f) \triangleq |M(f)|$ its cardinal. For example, the models of $v_1 \wedge (v_2 \vee \neg v_2)$ are $\{v_1 \mapsto \top, v_2 \mapsto \bot\}$ and $\{v_1 \mapsto \top, v_2 \mapsto \top\}$.

6.2 The f-E-MAJSAT Problem

For any propositional formula with free variables partitioned in two sets, f**-E-MAJSAT** consists in finding an assignment to variables in the first set maximizing the number of solutions for the other:

Definition 6.1 (f*-**E-MAJSAT** [Littman et al., 1998]).* Let $f \in \mathcal{F}$ and a partition of variables $A \uplus X = V(f)$.

$$\text{emajsat}_A(f) \triangleq \max_{a_1,\dots,a_n \in \mathbb{B}^A} \sharp\left(f|_{a_1,\dots,a_n}\right)$$

In the following, we will assume that all propositional formulas are expressed in *CNF*. As usual with functional problems, there is a companion decision problem called **E-MAJSAT** which tests whether f-**E-MAJSAT** is above a given threshold, *e.g.* $2^{|X|-1}$. Variables in A are called *choice variables* and variables in X are called *chance variables*. f-**E-MAJSAT** reduces to **SAT** when $X = \varnothing$ and to $\sharp$**SAT** when $A = \varnothing$ thus is at least as hard as these problems. **E-MAJSAT** is NP^{PP}-complete [Littman et al., 1998], meaning that it would become NP with a PP oracle.

6.3 Path-Wise Quantitative Robustness

The distinction between choice and chance variables mirrors the threat model of quantitative robustness, the former representing controlled inputs and the latter uncontrolled inputs. Assuming that path-constraints can be encoded as propositional formulas, we can thus conflate f-**E-MAJSAT** and path-wise quantitative robustness.

We represent inputs as boolean variables: $a \triangleq (a_1, \dots, a_n)$ and $x \triangleq (x_1, \dots, x_m)$. We define two formulas $h_a(a)$ and $h_x(x)$ specifying valid inputs: $\sharp(h_a) = |\mathcal{A}|$ and $\sharp(h_x) = |\mathcal{X}|$.

Proposition 6.2. *For a path constraint* pc_π^O *expressed as a propositional formula, path-wise quantitative robustness can be reduced to* f*-**E-MAJSAT** as follows:*

$$q(P|^\pi, O) = \frac{\text{emajsat}_a\left(h_a(a) \wedge h_x(x) \wedge \text{pc}_\pi^O(a,x)\right)}{\sharp(h_x)}$$

Path constraints are typically encoded as SMT formulas to benefit from the great expressiveness of theories supported by SMT solvers. However, it is possible to reduce some theories to SAT by bitblasting such that there is a unique corresponding model in the resulting formula for each one of the original, thus preserving model counts. More specifically, arrays can be eliminated by eager application of the read-over-write axiom of the theory and bitvectors can be bitblasted by mimicking the logical gates used in processors.

However, there is a major obstacle to the practical applicability of Proposition 6.2: solving f-**E-MAJSAT**.

7 EFFICIENT APPROXIMATION OF f-E-MAJSAT

In this section we turn to the problem of solving f-**E-MAJSAT** on a bitblasted path constraint. As quantitative robustness is only a hint for one dimension of exploitability, approximate methods are acceptable, yet efficiency is paramount to ensure practical applicability. Existing methods turn out to be either precise but too slow, or too coarse. We propose a novel approximation method that gives good results in practice.

7.1 Preliminary: Decision-DNNF Normal Form

Several techniques to solve f-**E-MAJSAT** rely on a normal form called *decision Decomposable Negational Normal Form (decision-DNNF)* [Fargier and Marquis, 2006].

Definition 7.1 (decision-DNNF). A formula in *decision-DNNF* is a DAG with the following nodes:

True and False nodes. $\top$ and $\bot$.

Decomposable And node. $\bigwedge_{i=1}^n f_i$, where for $1 \le i, j \le n$, $V(f_i) \cap V(f_j) = \varnothing$ and the children $(f_i)_{1 \le i \le n}$ are in *decision-DNNF*.

Decision or Ite ("if-then-else") node. $\mathrm{ite}(v, f, g)$, where f and g are formulas in *decision-DNNF* and v is a variable such that $v \notin V(f), v \notin V(g)$.
If additionally $V(f) = V(g)$ then the formula is said to be *smooth*.

This definition is slightly non-standard: literals are normally included, but we replace v by $\mathrm{ite}(v, \top, \bot)$ and $\neg v$ by $\mathrm{ite}(v, \bot, \top)$.

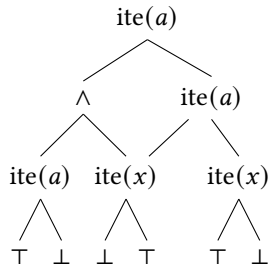


Fig. 3. a *decision-DNNF*

An example is given in Figure 3. By convention, $V(\top) = V(\bot) = \varnothing$, $\sharp(\top) = 1$, $\sharp(\bot) = 0$. For smooth Ite nodes, we have $\sharp(\mathrm{ite}(v, f, g)) = \sharp(f) + \sharp(g)$. Without smoothness, one must reason about pairs $(\sharp(f), V(f))$ instead of $\sharp(f)$, rendering the formal treatment considerably heavier. As per usual in the literature, we present the formalism on smooth formulas only, which does not incur a loss of generality [Darwiche, 2000] since a formula can be made smooth in polynomial time.

Model Counting and Compilation. Model counting of a formula in *decision-DNNF* can be done in linear time [Darwiche, 2001] (the algorithm is a special case of Definition 7.3). This reduces model counting to the process of converting a CNF formula to an equivalent *decision-DNNF* formula, which is called *compilation*. D4 [Lagniez and Marquis, 2017], C2D [Darwiche, 2004] and Dsharp [Muise et al., 2012][3] are examples of *decision-DNNF* compilers.

Compilation is significantly more expensive than model counting: it comprises roughly 96% of runtime on our test suite (see Section 8.3).

For a partial valuation $a \in \mathbb{B}^A$ and a formula f in *decision-DNNF* it is possible to compute $f|_a$ also in *decision-DNNF* as follows: replace $\mathrm{ite}(v, g, h)$ by g if $v \in A$ and $a(v) = \top$, h if $v \in A$ and $a(v) = \bot$, otherwise do nothing. We can thus compute $\sharp\left(f|_a\right)$ in linear time.

Layering. For f**-E-MAJSAT** on *decision-DNNF* formulas, one needs an extra constraint compared to model counting:

Definition 7.2. A formula in *decision-DNNF* is (A, X)-layered if $V(f) \subseteq A \uplus X$ and for any Ite node $\mathrm{ite}(v, f, g)$, we have $v \in X \implies V(f) \subseteq X$.

In other words, Ite nodes on variables in A are on top and those on variables in X below. This is the case on Figure 3 with $A = \{a\}$ and $X = \{x\}$.

Some *decision-DNNF* compilers such as Dsharp [Muise et al., 2012] can produce layered *decision-DNNF* as it can be used for projected model counting [Lagniez and Marquis, 2019], but this is significantly more expensive than unconstrained (non-layered) compilation.

[3]The latter two officially output a looser normal form called *deterministic Decomposable Negational Normal Form (d-DNNF)* [Darwiche, 2001] but actually produce *decision-DNNF*.

7.2 Preliminary: Solving f-E-MAJSAT with Decision-DNNF Normal Form

One straight-forward way to solve f-**E-MAJSAT** using layered *decision-DNNF* is by mapping And nodes to multiplication, chance Ite nodes to addition and choice Ite nodes to maximum [Huang, 2006, Pipatsrisawat and Darwiche, 2009]. This algorithm is known as Constrained. It also produces a *witness valuation, i.e.* a valuation a such that $Constrained(f) = \sharp\left(f|_a\right)$.

Definition 7.3 (Constrained algorithm [Huang, 2006]). For f in $(A, \mathcal{V} \setminus A)$-layered smooth *decision-DNNF*, we defines the count $C(f)$ and the witness valuation $w_A(f)$ as follows:

f	$C(f)$	$w_A(f)$
$\top$	1	$a_\bot$
$\bot$	0	$a_\bot$
$\text{ite}(v, g, h)$ with $v \notin A$	$C(g) + C(h)$	$a_\bot$
$\text{ite}(v, g, h)$ with $v \in A$	$\max(C(g), C(h))$	$w_A(h)\,[v := \bot]$ if $C(g) < C(h)$
		$w_A(g)\,[v := \top]$ otherwise
$\bigwedge_{i=1}^n g_i$	$\prod_{i=1}^n C(g_i)$	$w_A(g_1) \,\|\|\, \ldots \,\|\| w_A(g_n)$

With $a_\bot$ the partial valuation where all variables in A are mapped to $\bot$, $a[v := x]$ the valuation mapping v to x and other variables v' to $a(v')$, and $a||b$ the concatenation of valuations a and b.

Proposition 7.4. $Constrained(f) = \text{emajsat}_A(f)$.

In practice, this algorithm performs poorly due to the cost of constrained compilation. Note that Definition 7.3 can be used on a non-layered formula, yielding an upper bound [Huang, 2006]. We call this algorithm Unconstrained.

Proposition 7.5. $Unconstrained(f) \geq \text{emajsat}_A(f)$.

Other algorithms such as Complan [Huang, 2006] and Complan+ [Pipatsrisawat and Darwiche, 2009] improve this upper bound using various techniques. In particular, the latter uses a different upper bound, which we refer to as Oval. As we will see in our experimental evaluation in Section 8.3, the cost of constrained compilation renders algorithms like Constrained too expensive, while upper bounds like Oval based on unconstrained compilation are too loose to be applicable for quantitative robustness.

7.3 Our Proposition: Relaxation

We now propose an algorithm combining the advantages of Constrained (precision) and Oval (performance). We achieve this by relaxing the layering constraint on *decision-DNNF* compilation. Specifically, we compile $(A \uplus R, X \setminus R)$-layered *decision-DNNF* instead of (A, X)-layered, with R meant to be small. This allows the compiler to decide on $A \cup R$ instead of just A.

Upper Bound. We adapt Unconstrained to obtain an upper bound on $\text{emajsat}_A(f)$.

Definition 7.6 (Relaxed upper bound). Let f be a formula in $(A \uplus R, X)$-layered smooth *decision-DNNF*. We define $Relax^+(f) \in \mathbb{N}$ inductively as follows:

f	$Relax^+(f)$
$\top$	1
$\bot$	0
$\text{ite}(v, g, h)$ with $v \in X \cup R$	$Relax^+(g) + Relax^+(h)$
$\text{ite}(v, g, h)$ with $v \in A$	$\max(Relax^+(g), Relax^+(h))$
$\bigwedge_{i=1}^n g_i$	$\prod_{i=1}^n Relax^+(g_i)$

Proposition 7.7. $Relax^+(f) \geq \text{emajsat}_A(f)$.

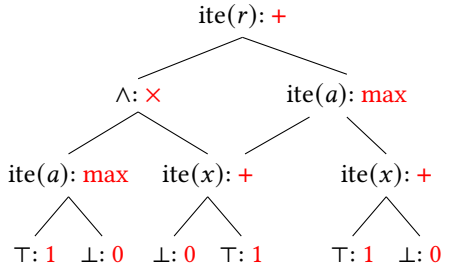
Proof. We prove the result by induction on the structure of f.

Computing $Relax^+(g)$ for g in the lower layer of f coincides with computing $\text{emajsat}_{A\cup R}(g)$ in Definition 7.3, but since $V(g) \cap R = \varnothing$, $Relax^+(g) = \text{emajsat}_{A\cup R}(g) = \text{emajsat}_A(g)$.

In the case of $Relax^+(f_A)$, where $f_A = \text{ite}(v, g, h)$ with $v \in A$, $\text{emajsat}_A(f_A) = \max(\text{emajsat}_A(g), \text{emajsat}_A(h))$. By induction hypothesis, $\text{emajsat}_A(g) \leq Relax^+(g)$ and $\text{emajsat}_A(h) \leq Relax^+(h)$. As max is non-decreasing in both its arguments, we prove the desired result $\text{emajsat}_A(f_A) \leq \max(Relax^+(g), Relax^+(h))$.

The same reasoning applies for the product on decomposable And nodes.

The case of a relaxed Ite node is the most interesting one: $f_R = \text{ite}(v, g, h)$, where $v \in R$. As $v \wedge g$ and $\neg v \wedge h$ have no common model, $M(f_R) = M(v \wedge g) \uplus M(\neg v \wedge h)$. Therefore, for a partial model $a \in \mathbb{B}^A$, we have $\sharp\left(f_R|_a\right) = \sharp\left((v \wedge g)|_a\right) + \sharp\left((\neg v \wedge h)|_a\right) = \sharp\left(g|_a\right) + \sharp\left(h|_a\right) \leq \text{emajsat}_A(g) + \text{emajsat}_A(h)$. Hence, $\text{emajsat}_A(f_R) \leq \text{emajsat}_A(g) + \text{emajsat}_A(h)$. By induction hypothesis $\text{emajsat}_A(g) \leq Relax^+(g)$ and $\text{emajsat}_A(h) \leq Relax^+(h)$, and thus $\text{emajsat}_A(f_R) \leq Relax^+(h) + Relax^+(g)$. □



For $f = \text{ite}(r, a \wedge \neg x, \text{ite}(a, \neg x, x))$, Proposition 7.7 yields $\text{emajsat}_{\{a\}}(f) \leq \max(1, 0) \times (0 + 1) + \max(0 + 1, 1 + 0) = 2$.

Fig. 4. $(\{a, r\}, \{x\})$-layered *decision-DNNF* (black), with Relax upper bound for it (red, Definition 7.6).

The principle is the same as Unconstrained except that relaxed choice Ite nodes map to addition like chance Ite nodes. An example is given in Figure 4.

Lower Bound. With Constrained, we can compute a witness $w_{A\cup R}(f)$ for $\text{emajsat}_{A\cup R}(f)$ in linear time (Definition 7.3). As its model count is maximal for $A \cup R$[4], we can expect it to have a good model count when restricted to A.

Definition 7.8 (Lower bound). Let f a formula in $(A \uplus R, X)$-layered smooth *decision-DNNF*. Let $w \in \mathbb{B}^A$ be a partial valuation coinciding with $w_{A\cup R}(f)$ for variables in A.

$$Relax^-(f) \triangleq \sharp\left(f|_w\right)$$

Proposition 7.9. $Relax^-(f) \leq \text{emajsat}_A(f)$.

Quality of the Resulting Interval. We propose Relax, the following algorithm:

Definition 7.10 (Relax). Let f be a formula in *CNF*, $A \uplus X$ a partition of its variables and $R \subseteq X$. Let f' be the compilation of f to $(A \uplus R, X \setminus R)$-layered *decision-DNNF*.

$$\text{Relax}(f) = [Relax^-(f'), Relax^+(f')]$$

[4] $\sharp\left(f|_{w_{A\cup R}(f)}\right) = \text{emajsat}_f(A \cup R)$

Computing the interval is done in linear time in the size of the *decision-DNNF*. The main parameter of Relax is R the set of relaxed variables. R is meant to be small enough to give good approximation, but large enough to allow tractable compilation. In the edge case where R is empty (no relaxation), the algorithm degenerates to Constrained and the resulting interval becomes a singleton. Conversely, when $R = X$, the algorithm degenerates to Unconstrained.

In general, we guarantee the following:

Theorem 7.11 (Precision of Relax). $Relax^+(f) \leq 2^{|R\cap V(f)|} Relax^-(f)$

Proof. With notation $L(f) \triangleq \text{emajsat}_{A\cup R}(f)$.

First we prove that $Relax^-(f) \geq L(f)$. Let $w \in \mathbb{B}^A, w' \in \mathbb{B}^R$ such that $w_{A\cup R}(f) = w||w'$. Each model x of $f|_{w_{A\cup R}(f)}$ can be mapped to a model $w'||x$ of $f|_w$. Therefore, $f|_{w_{A\cup R}(f)}$ has fewer models than $f|_w$, which can be written as $L(f) \leq Relax^-(f)$.

Then we prove $Relax^+(f) \leq 2^{|R\cap V(f)|} L(f)$ *by induction.* We compare rules in Definition 7.6 and Definition 7.3.

For base cases $\top$ and $\bot$, $Relax^+(f) = L(f)$.

For an Ite node with variable in X, $Relax^+(f) = L(f) = \sharp(f)$ and $R \cap V(f) = \varnothing$, by layering hypothesis.

In the case of an And node $f = \bigwedge_{i=1}^n g_i$, $Relax^+(f) = \prod_{i=1}^n Relax^+(g_i) \leq \prod_{i=1}^n 2^{|R\cap V(g_i)|} L(g_i) = \prod_{i=1}^n 2^{|R\cap V(g_i)|} \times \prod_{i=1}^n L(g_i) = 2^{\sum_{i=1}^n |R\cap V(g_i)|} L(f)$. Given that $V(f) = \biguplus_{i=1}^n V(g_i)$ we get $Relax^+(f) = 2^{|R\cap V(f)|} L(f)$.

For an Ite node $f = \text{ite}(v, g, h)$ with $v \in A$, $Relax^+(f) = \max(Relax^+(g), Relax^+(h)) \leq \max(L(g), L(h)) = L(f)$.

For a relaxed Ite node, $f = \text{ite}(v, g, h)$ with $v \in R$. $Relax^+(f) = Relax^+(g) + Relax^+(h)$. By induction hypothesis, $Relax^+(g) \leq 2^{|R\cap V(g)|} L(g) = 2^{|R\cap V(f)\setminus\{v\}|} L(g)$ and similarly for h. Finally, $Relax^+(f) \leq 2^{|R\cap V(f)\setminus\{v\}|} (L(g) + L(h)) \leq 2^{|R\cap V(f)\setminus\{v\}|} \times 2\max(L(g), L(h)) = 2^{|R\cap V(f)|} \max(L(g), L(h)) \leq 2^{|R\cap V(f)|} L(f)$. □

In practice, we show in Section 8 that imprecision is often lower than this.

Conclusion. Relax (Definition 7.10) is a parametric algorithm bridging the gap between Constrained and Unconstrained: the less relaxed variables there are, the more precise the answer, but the steeper the computational price.

8 IMPLEMENTATION & EXPERIMENTS

We first describe our implementations of *QRSE* (Binsec/QRSE) and f**-E-MAJSAT** solving (Popcon), then we evaluate the practical feasibility and relevance of the ideas developed so far.

8.1 Binsec/QRSE

We modified the binary-level robust symbolic execution engine BINSEC/RSE [David et al., 2016, Djoudi and Bardin, 2015, Girol et al., 2021] to perform *QRSE* on executable programs, using Popcon (see Section 8.2) as a f**-E-MAJSAT** solver. For optimization, the solver is only used for locations which are reachable (standard *SE* queries) but not robustly reachable (*RSE* queries). We also benefit from BINSEC optimizations, such as heavy array preprocessing [Farinier et al., 2018b].

Our tool only supports uniform distributions for uncontrolled inputs, but it is possible to specify their domain as intervals and with free-form assumptions. For example, it allows specifying *Address Space Layout Randomization (ASLR)* for the initial value of the stack register *esp* as $esp \in [\texttt{0xaaaa}, \texttt{0xbbbb}]$ and **assume** $esp\%16 = 0$ (alignment).

8.2 Popcon, a Front-End for f-E-MAJSAT Algorithms

For these experiments we implemented Popcon, a front-end for f-**E-MAJSAT** solvers accepting SMTLib2(QF_BV) and DIMACS input. It transparently converts this input to an appropriate format for the selected algorithm, including bitblasting with Boolector [Niemetz et al., 2015] (array blasting is done in BINSEC when necessary), and defers to an existing f-**E-MAJSAT** solver or a reimplementation when not available. Popcon consists in about 8k lines of Rust.

decision-DNNF-based algorithms (Oval, Constrained, and Complan+, see Section 6.2) are reimplemented, and compilation is performed using D4 [Lagniez and Marquis, 2017]. As Oval only provides an upper bound, we combine it with the lower bound from Section 7.3.

Popcon can also submit the formula to solvers based on different principles: (1) dc-ssat [Majercik and Boots, 2005] is a solver for probabilistic planning problems with arbitrarily many SSAT [Papadimitriou, 1985] quantifier alternations. We use a patched version with a different input format kindly provided by N.-Z. Lee; (2) ssatABC [Lee et al., 2018] is a solver for 2-quantifier SSAT problems based on clause selection; (3) Maxcount [Fremont et al., 2017] is an approximate, probabilistic solver for Max#SAT. Note that these solvers are not explicitly designed for f-**E-MAJSAT** but for more general problems.

Relaxation. Popcon provides an implementation of Relax (Section 7.3). This is achieved by querying D4 for a $(A \uplus R, X)$-layered *decision-DNNF* formula instead of a $(A, R \uplus X)$-layered one. Popcon offers two ways to choose R under the constraint that $|R| \leq r$, where r is a user-controlled parameter:

- **DFS**(r)**:** Starting with $R = \varnothing$, we patch D4 to add variables it would have decided if not constrained to R until $|R| = r$. R thus contains the first r variables the compiler tries to decide. D4 operates in depth-first search order, hence the name;
- **BFS**(r)**:** In this mode we mimic the decisions of model counting by running D4 for model counting and collecting the r top-most decided variables in breadth-first-search order in the resulting decision tree.

8.3 Research Questions

We aim at providing a first feasibility evaluation of the ideas developed so far. We consider the following research questions:

- **RQ1** Is quantitative robustness more precise than reachability and robust reachability?
- **RQ2** Can we find real examples where *QRSE* does not need path merging, while *RSE* does?
- **RQ3** How does Relax perform compared to state-of-the-art f-**E-MAJSAT** solvers?

8.4 Comparison with Robust Symbolic Execution (RQ1, RQ2)

Let us first compare the results of *RSE* and *QRSE* (with *RSE+* and *QRSE+* the respective variants with path merging) on Girol et al.'s original benchmark [Girol et al., 2021]. This benchmark is composed of both synthetic and real-world examples, including the *five real-world vulnerabilities* described in Table 1, some with multiple variants. The large discrepancy between the numbers of visited and unique instructions in some cases is explained by the need to explore long execution paths. The remaining programs include 4 CTFs and 8 library functions, which constitute synthetic scenarios on real code, as well as 26 toy examples. While most of these are relatively small (< 1k visited instructions), a few are on par with the CVEs (up to 100k visited instructions).

The ground truth for the experiment is whether robustness is above 20% and it is run with a timeout of 1 hour.

Table 1. Real-world vulnerabilities from Girol et al.'s benchmark [Girol et al., 2021]

Vulnerability		Instructions	
CVE	program	visited	unique
CVE-2019-19307	mongoose	165	89
CVE-2019-15900	doas	~ 1k	~ 100
CVE-2015-8370	grub	5k	137
CVE-2019-14192	uboot	32k	256
CVE-2019-20839	libvncserver	~ 40k	147

Table 2. Results of robust reachability analysis methods on Girol et al.'s benchmark [Girol et al., 2021] for proving quantitative robustness > 20%

Method	Results			Runtime (s)
	OK	FN*	T**	(total)
RSE	37	9	2	11765
RSE+	40	6	2	10012
QRSE	47	0	1	9348
QRSE+	46	0	2	10551

*false negatives **timeouts (1 hour)

Discussion Relative to RQ1. As shown in Table 2, the main advantage of *QRSE* over *RSE* in terms of precision is its ability to distinguish instances of high-yet-not-full robustness (> 20%) from those of low robustness, with the former shown as false negatives for *RSE* and *RSE+* in the results (9 and 6 occurrences here, where both variants of *QRSE* have no false negatives). We complement these results in Section 8.5 with a realistic case study where the precision improvements of *QRSE* over *RSE* does matter.

Discussion Relative to RQ2. Table 2 also shows that the lack of path merging with *QRSE* does not induce incorrect results in any case, i.e., there is always a path with at least 20% robustness when overall robustness exceeds that threshold. This is interesting as path merging is a potential source of constraint solving overhead and time out. For example, in comparison, *RSE* (without path merging) incurs more false negatives than *RSE+* (with path merging). In particular, we detail the case of one variant of libvncserver where *RSE* is wrong in Section 8.6.

QRSE also terminates on one more libvncserver variant compared to other methods. This illustrates how forgoing path merging can be beneficial in real-world applications.

8.5 Case Study on VerifyPIN [Dureuil et al., 2016] (RQ1)

We complete our answer to **RQ1** with a case study on vulnerability-oriented bug triage in the context of physical fault injection. We consider attackers controlling part of the input and able to inject a limited number of faults during the program execution. As other the combination of input choice and faults can be extremely complicated for a human to handle, finding potential attack traces needs to be partly automated.

This scenario is typical for the evaluation of highly sensitive hardware components such as smartcards, which must resist sophisticated intrusive attacks such as physical fault injection by mean of laser beams or electromagnetic waves. Since attempting such attacks can be time consuming and expensive, security evaluation often also consists in software-level analysis with (simulated) fault models. However potential attack traces must still be examined by a human to evaluate their exploitability. *Our goal is to reduce the amount of manual work needed by limiting the number of traces sent to experts, while still discovering all the most dangerous ones.*

Target. We consider the VerifyPIN_2 program from FISSC [Dureuil et al., 2016], a standard benchmark from the physical fault injection community [Giraud and Thiebeauld, 2004]. It is a procedure mimicking a typical password checker, including security-related countermeasures. It has two explicit inputs: the 4-byte entered PIN code *userPIN* and the secret PIN code stored on the card *cardPIN*.

While this program is relatively small, with only 60 unique instructions, the complexity resides in exploring all possible applications of the chosen fault model [Bardin et al., 2023, Lacombe et al.,

2023]. It is also representative of the size of code snippets considered in the fault analysis literature with powerful fault models.

Threat Model. We assume that attackers control *userPIN* but cannot influence *cardPIN* nor any uninitialized memory. Regarding faults, we assume that attackers are able to prevent the execution of single instructions, *i.e.* effectively replacing them with nop. The question is: *"Can such attackers enter a PIN distinct from the cardPIN and still be granted access?"*.

Methodology. We apply the 126 possible 1-byte and 2-byte wide nop faults on VerifyPIN, resulting in 126 *mutants*, i.e. individual programs including simulated faulty behaviour. We then use symbolic execution to find potential attacks (vulnerable mutants) and sort them according to quantitative robustness. This approach is fairly standard in fault analysis [Berthomé et al., 2012], although all-symbolic analysis is also possible [Bardin et al., 2023, Potet et al., 2014].

We compare the four following approaches (with a time out per query of 3min):

SE as implemented in BINSEC [Djoudi and Bardin, 2015];

RSE as implemented in BINSEC/RSE [Girol et al., 2021];

exact *QRSE* with Constrained, the most effective exact algorithm as shown in Section 8.7;

approximate *QRSE* with Relax, the best approximate algorithm as shown in Section 8.7, with the bfs choice heuristic. To get the best possible answer, we first try with 8 relaxed variables (tighter bounds) for half the allowed runtime, then with 128 (more consistent).

We distinguish traces which exhibit quantitative robustness above 0.2 (highly concerning) or below 10^{-6} (noise). For relaxed *QRSE*, we report traces *provably* in one of these categories, which are chosen to illustrate two approaches: a *conservative* analysis where only traces with a provably low quantitative robustness are dismissed and a more *optimistic* one where one only analyzes traces with high quantitative robustness.

Table 3. Comparison of various methods searching for fault attack traces with high robustness

Method	Quantitative robustness	Attack traces (found / truth)	Total runtime (s)	Timeouts (3min)
SE	$> 0\%$	39/39	66	–
RSE	$= 100\%$	0/0	67	–
exact *QRSE*	$> 20\%$	0/2	2435	13
	$< 10^{-6}$	23/27		
	$\in [10^{-6}, 20\%]$	3/10		
relaxed *QRSE* BFS(8) then BFS(128)	$> 20\%$	2/2	250	**0**
	$< 10^{-6}$	27/27		
	$\in [10^{-6}, 20\%]$	10/10		

Results. As shown in Table 3, *SE* reports 39 attack traces while *RSE* reports none, i.e. no perfect attack is found. On the other hand, quantitative approaches reports an intermediate number depending on the threshold. Exact *QRSE* has 13 timeouts, but still proves that out of the 39 attacks found by *SE*, at least 23 are not interesting ($< 10^{-6}$).

Relaxed *QRSE* is a significant improvement: it never times out while classifying 27 traces as not interesting and finding two with high quantitative robustness (~ 0.99 both). Manual analysis on the traces confirms these results. For example, the lowest quantitative robustness ($\sim 2^{-56}$) corresponds

to a case where attackers must guess 3 bytes of the cardPIN, the low byte of a register and hope for the top 3 bytes to be zero. Overall this amounts to 7 bytes, or 56 bits, of luck. On the other hand, the top 6 detected faults are outside the protected code of VerifyPIN, which proves that the protected part admits no attack with quantitative robustness above 10^{-4} within our threat model.

Conclusion for RQ1. In this case study, *QRSE* reduced the number of traces to analyze manually from 39 with standard *SE* to 12 in the most conservative scenario and 2 in the most optimistic one. Meanwhile, *RSE* does not highlight any cases as there are no 100% robust attack traces. Thus here *QRSE* helps to focus the attention of security experts on high robustness attack traces better than *RSE*, as expected from our initial comparison in Section 8.4. *Overall, quantitative robustness is a much more precise measurement of robustness than robust reachability, which is useful in practice as we just illustrated.*

8.6 Case Study on CVE-2019-20839 (RQ2)

We illustrate the benefits of the absence of path merging in a case study on **CVE-2019-20839** from `libvncserver` [lib, [n. d.]], an open-source library implementing remote computer-sharing utilities. The name of a socket file is copied to a fixed size stack buffer without checking its length, resulting in a stack buffer overflow potentially allowing attackers to overwrite stack data. The security question is: *Can attackers controlling this socket file's name divert control flow to* `0xdeadbeef`*?*

Standard *SE* tells us it is possible when the top of the stack is at `0xfff02000` and various other initial conditions are met. But most of those are beyond the control of the attacker, meaning this information is of little use for vulnerability assessment.

RSE can prove the stronger robust reachability: by choosing the right input, attackers can trigger the buffer overflow for all initial conditions. However, this requires systematic path merging, which can be useful when used carefully but detrimental to performance when used systematically [Hansen et al., 2009, Kuznetsov et al., 2012].

As explained in Section 5.2, path merging is not needed in *QRSE* when only few paths would need to be merged. Instead, we can attempt to detect single paths with high quantitative robustness. On this example, *QRSE* is indeed able to find a single path with quantitative robustness above 20%. The evidence is weaker than full robust reachability but still a good hint for security.

Conclusion for RQ2. Here *QRSE* is still able to find a single path with high robustness, which guarantees that overall robustness is at least equal, while *RSE+* is required to prove robust reachability. The same issue occurs for the other instances of additional false negatives for *RSE* compared to *RSE+* in Table 2 (Section 8.4). This reliance on path merging for *RSE* can also cause scalability issues, as we saw that *QRSE* is the only algorithm to terminate on another variant of `libvncserver`. *QRSE (without path merging) can give a strong indication of robustness by finding a single highly-robust path, while RSE relies much more on path merging.*

8.7 Performance of the Relax Solver (RQ3)

To answer the remaining research questions, we prepared a benchmark composed of 117 f-**E-MAJSAT** instances: 92 SMTLib2 formulas obtained from the original *RSE* paper Girol et al. [2021], including the real-world vulnerabilities detailed in Table 1; and 25 SMTLib2 problems generated in our VerifyPIN case study (**RQ1**).

The size of these formulas (554 variables and 998 clauses on average after bitblasting) is comparable to what is found in Lee et al. [2018] (331 variables and 3761 clauses on average). Problems are run on an Intel Xeon E-2176M CPU (2.70GHz) with a timeout of 20 minutes and a memory-out of 2 gigabytes.

Table 4. Runtime of solvers

Algorithm	Average (s) ok**	Average (s) all***	Timeouts*
Exact			
constrained	0.6	25.5	9/117
ssatabc	N/A	1200	**117/117**
complan+	23.7	1190	**116/117**
dcssat	10.8	79	**57/117**
Approximate			
oval	0.3	9.2	1/117
maxcount	N/A	1097	**117/117**
relax_bfs(8)	1.3	14	2/117
relax_dfs(8)	1.5	7	2/117
relax_bfs(32)	0.5	12.6	2/117
relax_dfs(32)	1	7.7	2/117
relax_bfs(128)	0.3	9.4	1/117
relax_dfs(128)	0.2	4.5	1/117

* 20 minutes ** without timeouts *** with timeouts

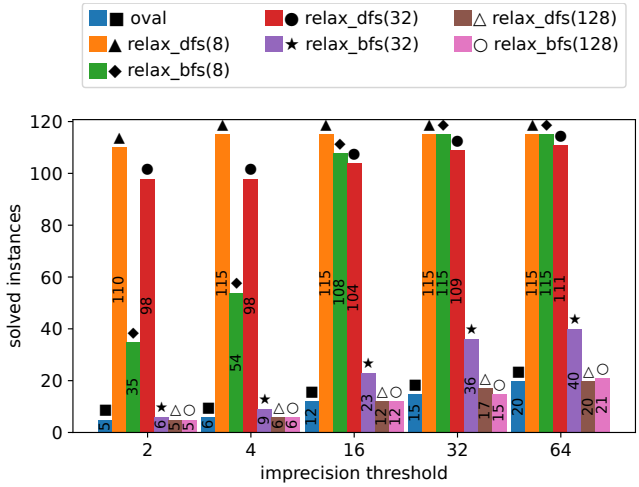


Fig. 5. Number of instances solved by approximate algorithms under a precision threshold

Exact Methods. Only two exact methods can solve a significant number of instances as shown in Table 4: DC-SSAT (60/117) and CONSTRAINED (108/117). The performance of DC-SSAT is still quite poor here, with a high average runtime even without timeouts. However CONSTRAINED performs surprisingly well for what is supposed to be an impractical, naive baseline, although it still times out 9 times, which significantly impacts performance in practice. This could be due to problems stemming from program analysis differing significantly from those for which COMPLAN+, SSATABC and DC-SSAT are designed. Approximate algorithms are thus needed to solve more than 108/117 instances, with less timing-out.

Approximated Methods. In order to fairly evaluate approximate algorithms, we must consider both their time performance and the precision of their result. Let l and h be the lower and upper bounds returned by approximate solvers. We call *imprecision* the ratio h/l.

Table 4 shows that OVAL terminates on 116 instances with good average runtime. However these numbers are misleading as precision is not taken into account. As we can see in Figure 5, OVAL's results are very imprecise, on par with RELAX's worst set up. Also, MAXCOUNT always times out here, likely for similar reasons as the poor performance of exact solvers.

The RELAX Algorithm. Table 4 shows that RELAX terminates on 115 to 116 instances depending on parameters, with low average runtime. Overall, RELAX with the *dfs* relaxed variable selection heuristic and 8 relaxed variables gives the most precise results as we can see on Figure 5: 115 instances are solved with an imprecision threshold of only 4. In particular, it solves 7 out of the 9 problems which CONSTRAINED cannot, while the latter fails to solve any that RELAX times out on. RELAX is still fairly precise with 32 relaxed variables, however 128 is clearly too many. See Appendix A for a more detailed study on the impact of the number of relaxed variables. Note that the *bfs* relaxed variable selection heuristic does not seem as good as *dfs* and that RELAX's results with 128 relaxed variables are on par with OVAL's.

Figure 6 gives a summary of the performance of all algorithms with an imprecision threshold of 32. CONSTRAINED is the best exact algorithm, however approximate approaches can solve more instances and RELAX is the best of them both in terms of solved instances and precision.

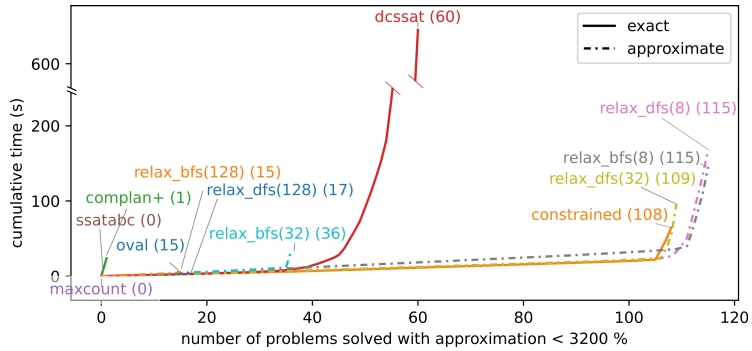


Fig. 6. Cactus plot of various f-**E-MAJSAT** solving algorithms applied to 117 instances from *QRSE*. Dashed lines correspond to methods returning an interval $[l, h]$ rather than an exact answer. Only solutions with imprecision h/l below 32× are considered solved. The number of solved instances is given in parentheses. Note that the y axis is broken to improve readability.

Conclusion for RQ3. *The* Relax *algorithm performs much better than prior works on the kinds of instances generated by QRSE, both in terms of speed (very few time outs) and precision. Additionally, the naïve* Constrained *algorithm performs surprisingly well.*

9 DISCUSSION

Choosing Controlled Inputs. In most instances, choosing controlled inputs should be relatively straightforward. However if that is not the case, a lower bound on quantitative robustness can still be obtained by under-approximating the set A of controlled inputs. Techniques such as dynamic analysis (taint propagation) can possibly be leveraged here to discover such under-approximations.

Uncontrolled Inputs Distribution. We assume that uncontrolled inputs are uniformly distributed, which may not be the case in practice. Thus, it is possible in principle for some uncontrolled inputs allowing or preventing a vulnerability to be more or less common than others, in which case *QRSE* results could be inaccurate. As a first mitigation, user-given assumptions on uncontrolled inputs can be taken into account within our method, preventing impossible or very rare uncontrolled input from being taken into account.

Also, while our formal framework could certainly be extended to take into account such input distribution, this would come with two significant practical issues. First, estimating such distribution in practice is certainly a challenge on its own, and the user would be left with this burden. Second, current model counting techniques do not take input distribution into account, hence we cannot currently automate such a feature. One way to solve this issue could be to take a probabilistic sampling approach, although this would come at the cost of weaker formal guarantees.

Finally, note that, to our knowledge, the closest method to *QRSE* from the literature, probabilistic symbolic execution [Geldenhuys et al., 2012], has the same limitation.

Handling Richer Theories. Since Binsec uses bitvector and array theories to build path constraints, we focused on those when designing our algorithms. However, handling other theories would be beneficial as it would simplify input specifications for example.

While some solutions have been proposed for handling theories such as string in standard model counting [Aydin et al., 2015, 2018, Trinh et al., 2017], the state-of-the-art for f-**E-MAJSAT** solvers has not reached that point yet. In addition, the model counting algorithms proposed there rely on

structures such as automata [Aydin et al., 2015, 2018] rather than bitblasting and compilation to *decision-DNNF*, making an adaptation non trivial.

Practical Usability. As shown in Table 2, the overall runtimes of *RSE* and *QRSE* on Girol et al.'s benchmark [Girol et al., 2021] are on par with each other. In general, *QRSE* should be at least as scalable as RSE, although the lesser reliance on path merging is a definite advantage. However all the shortcomings of symbolic execution still apply: path explosion, unmanageable constraints, high expertise requirements.

Still, *QRSE* may have a higher potential for optimization that *RSE* as high robustness paths can be preferentially explored, analysis halted when total robustness exceeds a given threshold, paths selectively merged, etc. There is a clear research direction for performance improvement here. In addition, it should combine better with traditional optimizations of symbolic execution, such as concolic execution [Sen et al., 2005] or under-constrained *SE* [Ramos and Engler, 2015]. Finally, QRSE could be applied to a few pre-selected paths as part of a broader evaluation process.

10 RELATED WORKS

10.1 Comparison to Other Quantitative Formalisms

We designed a quantitative counterpart to robust reachability, which we view as too strict. Quantitative relaxation has already been seen in other domains and is part of a general effort to make formal verification less "all-or-nothing": from non-interference [Goguen and Meseguer, 1982] to quantitative information flow [Heusser and Malacaria, 2010], from traditional model checking to probabilistic model checking [Aziz et al., 1996, Hansson and Jonsson, 1994] or from symbolic execution to probabilistic symbolic execution [Geldenhuys et al., 2012]. These different applications give rise to different counting or probabilistic problems. We rely on f**-E-MAJSAT** while probabilistic verification builds on standard model counting [Gomes et al., 2008], probabilistic model checking on Markov chains and quantitative information flow on projected model counting [Aziz et al., 2015].

Probabilistic Reachability. Program verification is usually encoded as the reachability of an undesirable condition, thus a natural evolution is to consider the probability of reaching it. For example probabilistic symbolic execution [Geldenhuys et al., 2012] attempts to compute the probability[5] of each path, and shows experimentally that one can find bugs by focusing human analysis on improbable paths. The main difference with our work is that they compute the probability of a bug happening in a neutral environment, whereas we take into account the presence of an attacker.

Probabilistic Temporal Logics. Probabilistic logics developed for model checking like pCTL [Hansson and Jonsson, 1994] use Markov chains instead of model counting on constraint systems. They can express the probability of complex events in interactive systems with several rounds of input, but not systems where two actors have different interests. Mapping the CTL encoding of robust reachability (**EXAF**φ) to pCTL expresses the probability of reaching the target property for a specific attacker whose probability transition tables are known. However, in our case attacker actions should be taken as worst case and are not known *a priori*. More expressive logics like MTL_2 [Jamroga, 2008], a generalisation of ATL [Alur et al., 2002], can express a worst-case attacker, but are so general that they lack tractable proof methods.

Quantitative Information Flow. Quantitative information flow attempts to quantify the amount of information that an attacker can deduce from the observable behavior of a system, interpreted as leakage of information. Attackers choose public inputs to a system, defenders choose secret inputs

[5]Note that they compute model counts and therefore also assume uniformly distributed inputs.

and attackers attempt to deduce the secret from the public output [Heusser and Malacaria, 2010]. A central notion is the capacity of the leakage channel: the logarithm of the number of public outputs z such that there exists a pair of (public, private) inputs leading to z. This problem is called *projected model counting* [Aziz et al., 2015] and is distinct from our approach based on f**-E-MAJSAT**.

Phan et al. [Phan et al., 2017] automatically build multi-step side-channel attacks maximizing information leakage. Since the leakage for a given set of observations (and thus the underlying execution paths) is constant, they fall under the MaxSMT problem, where the goal is to maximize the sum of weights corresponding to different satisfied constraints. In contrast, the weights of executions paths in the analogous quantitative robust reachability problem, i.e. the number of triggering uncontrolled inputs, depend on controlled inputs, hence our reduction to the harder f**-E-MAJSAT** problem.

10.2 Other Related Works

Counting Solvers. Many combinations and extensions are possible. The branch-and-bound algorithms behind Complan and Complan+ can be interrupted at any time to obtain a refined, but not perfect interval. Our algorithm Relax could be refined by using bounds inspired from Oval instead of Unconstrained, at the price of significant added complexity. The choice of the set of relaxed variables has only been partially explored, and is certainly a direction for future work. We could also get inspiration from a recently proposed CEGAR-based approach to counting [Vigouroux et al., 2022]. Note finally that some works target model counting beyond propositional formulas (e.g., for bit-vectors [Kim and McCamant, 2018] or integer polyhedra [De Loera et al., 2004]), which could guide us for further developments.

Flakiness. When a branch can be reached robustly, but outgoing paths are not robust anymore, then some dependence on uncontrolled input may be at play. If uncontrolled inputs are non-deterministic, this strongly suggests [Girol et al., 2021] that the test is *flaky* (has non-deterministic outcome), which is an active area of research [Alshammari et al., 2021, Luo et al., 2014, Wei et al., 2022]. Quantitative robustness could thus be used to detect locations where flakiness is introduced, in the form of branches with less quantitative robustness than their parent.

11 CONCLUSION

Robust reachability has been recently proposed for identifying bugs or attacks which are replicable with absolute certainty from those which are not. Yet, this qualitative robust reachability is too strong to distinguish mostly replicable bugs from unreplicable ones. We solved this issue by introducing the notion of Quantitative Robustness, together with *Quantitative Robust Symbolic Execution (QRSE)*, an analysis technique able to find buggy execution paths with high quantitative robustness. Especially, our approach reduces, in the finite domain case, to a variant of model counting named f**-E-MAJSAT**. Given the poor performance of existing f**-E-MAJSAT** solvers on quantitative robustness instances, we developed a new approximate algorithm, Relax. Our experiments show that quantitative robustness and *QRSE* can be useful in practice for the purpose of vulnerability evaluation.

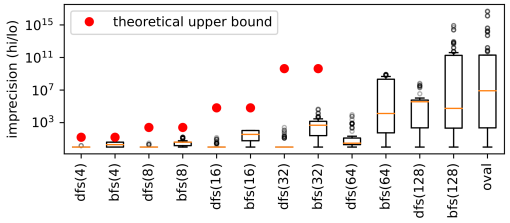
AVAILABILITY STATEMENT

All research artifacts and instructions for reproducing results are available at https://zenodo.org/records/10937863.

ACKNOWLEDGEMENTS

This work is supported by ANR, grant AAPG TAVA, and France Stratégie 2030, grants PEPR Cyber Secureval ANR-22-PECY-0005 and PEPR Cyber REV ANR-22-PECY-0009.

A IMPACT OF THE NUMBER OF RELAXED VARIABLES ON THE RESULTS OF RELAX



Theoretical upper bound (Theorem 7.11) 2^r omitted for $r \geq 64$.

Fig. 7. Box plot of imprecision (upper/lower bound) of approximate f-**E-MAJSAT** solving algorithms.

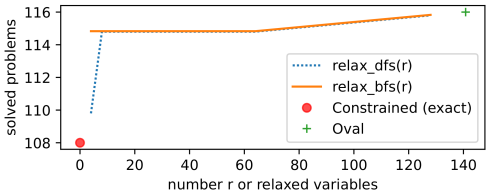


Fig. 8. Solved instances within timeout depending on the number r of relaxation variables, regardless of precision.

The number of instances solved by Relax within timeout increases with the number r of relaxed variables (Figure 8). Up to 8 more instances can be solved with relaxation. The imprecision also increases with r (Figure 7), although it is often orders of magnitude smaller than the theoretical bound 2^r (Theorem 7.11). DFS variable order usually yields more precise results, but for $r = 128$ this tendency is broken regarding the median. As expected, when r increases we observe similar behavior to techniques based on fully unconstrained *decision-DNNF*, like Oval. *Relaxation can reach a sweet spot between precision and efficiency, allowing to solve more instances than exact algorithms with significantly better approximation than theoretical bounds.*

REFERENCES

[n. d.]. https://github.com/LibVNC/libvncserver. Online, accessed November 17th 2023.

Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *Proceedings of the 43rd International Conference on Software Engineering*. IEEE Press, 1572–1584.

Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. 2002. Alternating-Time Temporal Logic. *J. ACM* 49, 5 (Sept. 2002), 672–713. https://doi.org/10/cgwb3h

Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-Based Model Counting for String Constraints. 255–272. https://doi.org/10.1007/978-3-319-21690-4_15

Abdulbaki Aydin, William Eiers, Lucas Bang, Tegan Brennan, Miroslav Gavrilov, Tevfik Bultan, and Fang Yu. 2018. Parameterized Model Counting for String and Numeric Constraints. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 400–410. https://doi.org/10.1145/3236024.3236064

Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. 1996. Verifying Continuous Time Markov Chains. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Rajeev Alur and Thomas A. Henzinger (Eds.). Springer, Berlin, Heidelberg, 269–276. https://doi.org/10.1007/3-540-61474-5_75

Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, and Peter Stuckey. 2015. #∃SAT: Projected Model Counting. In *Theory and Applications of Satisfiability Testing – SAT 2015 (Lecture Notes in Computer Science)*, Marijn Heule and Sean Weaver (Eds.). Springer International Publishing, Cham, 121–137. https://doi.org/10/gh6pzs

Sébastien Bardin, Soline Ducousso, and Marie-Laure Potet. 2023. Adversarial reachability for program-level security analysis. In *ESOP 2023 - 32nd European Symposium on Programming, (Programming Languages and Systems - proceedings of the 32nd European Symposium on Programming, Vol. 13990)*, ETAPS (Ed.). ETAPS, Springer, Paris, France, 58–89. https://doi.org/10.1007/978-3-031-30044-8_3

P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky, and J.-F. Lalande. 2012. High Level Model of Control Flow Attacks for Smart Card Functional Security. In *2012 Seventh International Conference on Availability, Reliability and Security*. 224–229. https://doi.org/10.1109/ARES.2012.79

Armin Biere, Alessandro Cimatti, Edmund Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded Model Checking. In *Advances in Computers*, Vol. 58. 117 – 148. https://doi.org/10.1016/S0065-2458(03)58003-2

Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795

Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Kurt Jensen, and Andreas Podelski (Eds.). Vol. 2988. Springer Berlin Heidelberg, Berlin, Heidelberg, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

Adnan Darwiche. 2000. On the Tractable Counting of Theory Models and Its Application to Truth Maintenance and Belief Revision. *Journal of Applied Non-Classical Logics* 11 (2000), 1–2.

Adnan Darwiche. 2001. Decomposable Negation Normal Form. *J. ACM* 48, 4 (July 2001), 608–647. https://doi.org/10/czk9nk

Adnan Darwiche. 2004. New Advances in Compiling CNF to Decomposable Negation Normal Form. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*. IOS Press, NLD, 318–322.

Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. 2016. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE Computer Society, 653–656.

Jesús A. De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. 2004. Effective Lattice Point Counting in Rational Convex Polytopes. *Journal of Symbolic Computation* 38, 4 (Oct. 2004), 1273–1302. https://doi.org/10/cf2mq7

Adel Djoudi and Sébastien Bardin. 2015. BINSEC: Binary Code Analysis with Low-Level Regions. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Christel Baier and Cesare Tinelli (Eds.). Springer, Berlin, Heidelberg, 212–217. https://doi.org/10.1007/978-3-662-46681-0_17

Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. 2016. FISSC: A Fault Injection and Simulation Secure Collection. In *Computer Safety, Reliability, and Security (Lecture Notes in Computer Science)*, Amund Skavhaug, Jérémie Guiochet, and Friedemann Bitsch (Eds.). Springer International Publishing, Cham, 3–11. https://doi.org/10/ggskcw

Hélène Fargier and Pierre Marquis. 2006. On the Use of Partially Ordered Decision Graphs in Knowledge Compilation and Quantified Boolean Formulae. In *Proceedings, the Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*. AAAI Press, 42–47.

Benjamin Farinier, Sébastien Bardin, Richard Bonichon, and Marie-Laure Potet. 2018a. Model Generation for Quantified Formulas: A Taint-Based Approach. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 294–313.

Benjamin Farinier, Robin David, Sébastien Bardin, and Matthieu Lemerre. 2018b. Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing. In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. 363–344. https://doi.org/10.29007/dc9b

Daniel Fremont, Markus Rabe, and Sanjit Seshia. 2017. Maximum Model Counting. *Proceedings of the AAAI Conference on Artificial Intelligence* 31, 1 (Feb. 2017).

Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic Symbolic Execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 166–176. https://doi.org/10/ggbn25

Christophe Giraud and Hugues Thiebeauld. 2004. A Survey on Fault Attacks. In *Smart Card Research and Advanced Applications VI (IFIP International Federation for Information Processing)*, Jean-Jacques Quisquater, Pierre Paradinas, Yves Deswarte, and Anas Abou El Kalam (Eds.). Springer US, Boston, MA, 159–176. https://doi.org/10/b5jk83

Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. 2021. Not All Bugs Are Created Equal, But Robust Reachability Can Tell the Difference. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, Cham, 669–693. https://doi.org/10/gmn5z6

Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. 2022. Introducing Robust Reachability. In *FMSD (CAV special issue)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer.

J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. IEEE, Oakland, CA, USA, 11–11. https://doi.org/10.1109/SP.1982.10014

Carla P. Gomes, Ashish Sabharwal, and Bart Selman. 2008. Model Counting. In *Handbook of Satisfiability*. IOS Press.

Trevor Hansen, Peter Schachte, and Harald Søndergaard. 2009. State Joining and Splitting for the Symbolic Execution of Binaries. In *Runtime Verification*, Saddek Bensalem and Doron A. Peled (Eds.). Vol. 5779. Springer Berlin Heidelberg, Berlin, Heidelberg, 76–92. https://doi.org/10.1007/978-3-642-04694-0_6

Hans Hansson and Bengt Jonsson. 1994. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing* 6, 5 (Sept. 1994), 512–535. https://doi.org/10.1007/BF01211866

Jonathan Heusser and Pasquale Malacaria. 2010. Quantifying Information Leaks in Software. In *Proceedings of the 26th Annual Computer Security Applications Conference on - ACSAC '10*. ACM Press, Austin, Texas, 261. https://doi.org/10.1145/1920261.1920300

Jinbo Huang. 2006. Combining Knowledge Compilation and Search for Conformant Probabilistic Planning. In *Proceedings of the Sixteenth International Conference on International Conference on Automated Planning and Scheduling (ICAPS'06)*. AAAI Press, Cumbria, UK, 253–262.

Wojciech Jamroga. 2008. A Temporal Logic for Stochastic Multi-Agent Systems. In *Intelligent Agents and Multi-Agent Systems (Lecture Notes in Computer Science)*, The Duy Bui, Tuong Vinh Ho, and Quang Thuy Ha (Eds.). Springer, Berlin, Heidelberg, 239–250. https://doi.org/10.1007/978-3-540-89674-6_27

Seonmo Kim and Stephen McCamant. 2018. Bit-Vector Model Counting Using Statistical Estimation. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 133–151. https://doi.org/10/ghtr84

Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. *SIGPLAN Not.* 47, 6 (June 2012), 193–204. https://doi.org/10.1145/2345156.2254088

Guilhem Lacombe, David Feliot, Etienne Boespflug, and Marie-Laure Potet. 2023. Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities. *Journal of Cryptographic Engineering* (Jan. 2023). https://doi.org/10.1007/s13389-023-00310-8

Jean-Marie Lagniez and Pierre Marquis. 2017. An Improved Decision-DNNF Compiler. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, Melbourne, Australia, 667–673. https://doi.org/10/gh6rkj

Jean-Marie Lagniez and Pierre Marquis. 2019. A Recursive Algorithm for Projected Model Counting. *AAAI* 33 (July 2019), 1536–1543. https://doi.org/10/ghkjdq

Nian-Ze Lee, Yen-Shi Wang, and Jie-Hong R. Jiang. 2018. Solving Exist-Random Quantified Stochastic Boolean Satisfiability via Clause Selection. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, Stockholm, Sweden, 1339–1345. https://doi.org/10.24963/ijcai.2018/186

M. L. Littman, J. Goldsmith, and M. Mundhenk. 1998. The Computational Complexity of Probabilistic Planning. *jair* 9 (Aug. 1998), 1–36. https://doi.org/10.1613/jair.505

Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 643–653. https://doi.org/10.1145/2635868.2635920

Stephen M. Majercik and Byron Boots. 2005. DC-SSAT: A Divide-and-Conquer Approach to Solving Stochastic Satisfiability Problems Efficiently. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1 (AAAI'05)*. AAAI Press, Pittsburgh, Pennsylvania, 416–422.

Christian Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu. 2012. Dsharp: Fast d-DNNF Compilation with sharpSAT. In *Advances in Artificial Intelligence (Lecture Notes in Computer Science)*, Leila Kosseim and Diana Inkpen (Eds.). Springer, Berlin, Heidelberg, 356–361. https://doi.org/10/gjjsfh

Aina Niemetz, Mathias Preiner, and Armin Biere. 2015. Boolector 2.0: System Description. *SAT* 9, 1 (June 2015), 53–58. https://doi.org/10/ghv4cd

Christos H. Papadimitriou. 1985. Games against Nature. *J. Comput. System Sci.* 31, 2 (Oct. 1985), 288–301. https://doi.org/10.1016/0022-0000(85)90045-5

Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of Adaptive Side-Channel Attacks. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. 328–342. https://doi.org/10.1109/CSF.2017.8

Knot Pipatsrisawat and Adnan Darwiche. 2009. A New D-DNNF-Based Bound Computation Algorithm for Functional E-MAJSAT. In *IJCAI*.

Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. 2014. Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections. In *Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014*. 213–222. https://doi.org/10.1109/ICST.2014.34

David A. Ramos and Dawson Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, USA, 49–64.

Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE Symposium on Security and Privacy*. 317–331. https://doi.org/10.1109/SP.2010.26

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE-13)*. Association for Computing Machinery, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2017. Model Counting for Recursively-Defined Strings. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 399–418.

Leslie G. Valiant. 1979. The Complexity of Enumeration and Reliability Problems. *SIAM J. Comput.* 8, 3 (1979), 410–421. https://doi.org/10.1137/0208032 arXiv:https://doi.org/10.1137/0208032

Thomas Vigouroux, Cristian Ene, David Monniaux, Laurent Mounier, and Marie-Laure Potet. 2022. BaxMC: a CEGAR approach to Max#SAT. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 170–178.

Anjiang Wei, Pu Yi, Zhengxi Li, Tao Xie, Darko Marinov, and Wing Lam. 2022. Preempting Flaky Tests via Non-Idempotent-Outcome Tests. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1730–1742. https://doi.org/10.1145/3510003.3510170

Received 2023-11-16; accepted 2024-03-31