Not all bugs are created equal, but robust reachability can tell the difference

Guillaume Girol, Benjamin Farinier, Sébastien Bardin

Not All Bugs Are Created Equal, But Robust Reachability Can Tell The Difference

Guillaume Girol[1], Benjamin Farinier[2], and Sébastien Bardin[1]

[1] Université Paris-Saclay, CEA, List, France first.last@cea.fr
[2] TU Wien, Vienna, Austria first.last@tuwien.ac.at

Abstract. This paper introduces a new property called *robust reachability* which refines the standard notion of reachability in order to take replicability into account. A bug is robustly reachable if a *controlled input* can make it so the bug is reached whatever the value of *uncontrolled input*. Robust reachability is better suited than standard reachability in many realistic situations related to security (e.g., criticality assessment or bug prioritization) or software engineering (e.g., replicable test suites and flakiness). We propose a formal treatment of the concept, and we revisit existing symbolic bug finding methods through this new lens. Remarkably, robust reachability allows differentiating bounded model checking from symbolic execution while they have the same deductive power in the standard case. Finally, we propose the first symbolic verifier dedicated to robust reachability: we use it for criticality assessment of 4 existing vulnerabilities, and compare it with standard symbolic execution.

1 Introduction

Context. Many problems in software verification are encoded as *reachability* queries of some undesired condition—a bug, the exploitation of a vulnerability, *etc.* When a verification engine establishes that a certain buggy location in the program is reachable, an input triggering the bug is reported to the developer so that it can be fixed. In the case of techniques based on an under-approximation of program behaviors, like Symbolic Execution (SE) [9] or Bounded Model Checking (BMC) [13], we even have *in principle* the guarantee that the reported issue is real (*correctness*): there are no false positives.

Problem. Yet, things are more subtle in practice, as some bugs can be triggered reliably whereas others only happen in very specific and highly improbable initial conditions. While standard reachability cannot tell the difference, this distinction is crucial in many real-life scenarios related to security (bug triage, bug prioritization, criticality assessment) or software engineering (test suite replicability and the problem of flaky tests [42]). For example, fuzzers are able to detect so many bugs [38] that they can lead to "bug triage issues" [30]. If each *replicable*

This work has been partially supported by ANR (grant ANR-20-CE25-0009-TAVA) and ERC (grant agreement 771527-BROWSEC).

(reliably-triggered) bug is hidden by dozens of more *fragile* ones in the reports of a verification engine, it is hard to focus development effort efficiently. Also, if one is only interested in vulnerability reports, bugs which cannot be reliably triggered may even be dismissed as "not exploitable" altogether.

Goal & challenges. *Our goal is to develop a formal framework able to distinguish replicable bugs from fragile bugs, and amenable to automatic software verification — precisely, we want to be able in practice to find such replicable bugs.* This is challenging as we need to avoid any quantitative [37] or probabilistic reasoning [34,2], insofar as they would hinder automation on real examples — these techniques are often either restricted to finite-state systems [34,2] or rely on highly expensive model counting solvers [39,11].

Proposal. Our approach consists in partitioning inputs of the program into *controlled inputs* and *uncontrolled inputs*. This lets us refine the concept of reachability into *robust reachability*: a (buggy) location of a program is robustly reachable if there exist controlled inputs, such that for all uncontrolled inputs, this location is reached. In other words, with adequate input we do not need luck.

We typically focus on *security* scenarios where an *attacker* provides controlled input in one go, without knowledge of uncontrolled input – typically sending a malicious crafted file to obtain remote code execution or privilege escalation. We *deliberately* exclude interactive attack scenarios and weaker interpretations like "bugs replicable *most of the time*" in order to keep proof methods *tractable*.

Proving robust reachability is harder than standard reachability. While we show that robust reachability is expressible in formalisms like branching temporal logics [14], hyperproperties [16] or hyper temporal logic [15], there exist no efficient automated analysis methods for these formalisms at the software level (for Turing-complete languages). Therefore, we investigate dedicated verification techniques, revisiting standard methods (SE, BMC) for standard reachability as well as some of their standard companion optimizations.

Our prototype of Robust Symbolic Execution (RSE) relies on the ability of state of the art Satisfiability Modulo Theory (SMT) solvers [4] to generate models for *universally* quantified formulas [27,25,44], which comes with a performance and completeness cost — yet we report promising results.

Contributions. We claim the following contributions.

- We formally introduce the concept of robust reachability (Sec. 4) and motivate its use (Sec. 2), giving practical examples where standard reachability leads to false positives in practice (whatever the underlying verification technology). We also characterize robust reachability in terms of temporal logic and hyperproperties, and compare it with *non-interference* (Sec. 4);
- We revisit Symbolic Execution (SE) [9] and Bounded Model Checking (BMC) [13] and show how they can be lifted to the robust case (Sec. 5). While they both have the same deductive power in the standard case, they do not anymore in the robust setting — yet, *path merging* allows Robust SE to pace up with Robust BMC. Finally, we show how to adapt standard optimizations for Symbolic Execution and Bounded Model Checking;

– We implement and evaluate[3] (Sec. 6) the *first* symbolic execution engine dedicated to robust reachability, namely BINSEC/RSE. We show how to use it for criticality assessment of 4 existing vulnerabilities (CVEs), and compare it with standard symbolic execution. RSE appears to be tractable with reasonable overhead, yielding false-positive-free symbolic reasoning.

We believe robust reachability is an important sweet spot in terms of expressiveness and tractability, allowing to highlight serious bugs in practical situations. We hope this first step will pave the way to more refinements and applications of robust reachability.

2 Motivation

In this section we show why standard reachability is not always a good fit for bug finding, as it cannot distinguish between *replicable* bugs and *fragile* bugs.

```
void fill(unsigned n, char* ptr) {
 for (unsigned i = 0; i < n; i++) {
   ptr[i] = 0x61;
 }
}
void victim() {
 unsigned n = controlled_input;
 char buffer[8];
 fill(n, buffer);
}
void main() {
 victim();
}
```

(a) C-like code, for simplicity

```
1  void victim() {
2   /* stack variables, top to bottom */
3   // return address goes here
4   int canary = global_random_value;
5   char buffer[8];
6   /* end stack variables */
7
8   register unsigned n = controlled_input;
9   fill(n, buffer);
10  if (canary != global_random_value)
11   fail_and_dont_return_at_all();
12  /* everything is ok */
13 }
```

(b) Explanation of compiler instrumentation with Stack Smashing Protection (SSP)

Fig. 1: Simple stack buffer overflow

Stack canaries. Consider the program presented in Fig. 1. It suffers from a stack buffer overflow: if variable `n` is greater than 8 (the size of `buffer`), then `0x61` will be written to stack memory above `buffer`. For high enough `n`, this will overwrite the return address (Fig. 1b, line 3) of function `victim` and make the program jump to an unexpected program location when `victim` returns.

Mitigations for such programming errors exist, like Stack Smashing Protection (SSP) [18]. This technique consists in pushing a randomly-chosen constant value called a *canary* at the top of the stack in the prologue of each function, and checking that this value is intact before returning. If the canary has been tampered with, the program exits to prevent exploitation (Fig. 1b, line 11). Here, SSP prevents the attacker from overwriting the return address of `victim`, as doing so also overwrites the canary with `0x61616161`. This will be detected at line 10 of Fig. 1b with probability $1 - 2^{-32}$ on a 32-bit architecture: the only way to pass through it is to have the canary value equal to `0x61616161`. *Hence, the buffer overflow in this program is not exploitable anymore.*

[3] The tool, benchmark and data are available at https://github.com/binsec/cav2021-artifacts and https://zenodo.org/record/4721753.

Table 1: Standard reachability is not a good criterion to measure the protection of SSP on the program of Fig. 1.

Prog. Fig. 1	Ground truth	Standard reachability	BINSEC [23]	Angr [46]	Robust reachability	BINSEC/RSE
No SSP	vulnerable	vulnerable ✓	vulnerable ✓	vulnerable ✓	vulnerable ✓	vulnerable ✓
SSP	protected	vulnerable ✗	vulnerable ✗	vulnerable ✗	protected ✓	protected ✓

The problem with standard reachability. Can the attacker hijack the control flow without triggering SSP? We can model this security question as a *standard reachability query* over inputs `controlled_input` and `global_random_value`. The attacker succeeds if line 12 is reachable with the additional condition that the return address of `victim` is overwritten with an unexpected address.

Unfortunately, this standard reachability query is satisfiable with the canary `global_random_value` equal to `0x61616161` and `controlled_input` equal to *e.g.*, 42. And indeed, binary-level SE tools Angr [46] or BINSEC [23] do report the bug as reachable (cf. Tab. 1). Yet, this answer is unsatisfying as this only happens with a very low probability: it may not be considered a plausible attack. *Hence, it turns out that SE can yield false positives in practice — especially in a security context.*

Proposal: robust reachability. We label `controlled_input` as a *controlled input* and `global_random_value` as an *uncontrolled input*. There exists no value of `controlled_input` such that `victim` returns to an address tampered with independently of the value of `global_random_value`. We thus say that our exploitation condition (line 12) is not *robustly reachable*. We can automatically verify this intuition. We adapted the SE engine of BINSEC to robust reachability: our tool finds the vulnerability when we disable the protection (by labelling the canary as *controlled input*) and does not find it anymore when the protection is present. This shows that robust reachability can model the protection provided by SSP, while standard reachability cannot.

This phenomenon is not restricted to stack protectors. We identify in Tab. 2 several situations where standard reachability may lead to false positives, unlike robust reachability. Note that some cases (randomisation based protections, uninitialized reads) concern binary-level issues, and cannot be observed from a source-level analysis.

Discussion. Consider the slightly different problem of reaching line 11 in Fig. 1b. It is reachable for all values of the canary *except* `0x61616161`, hence it is not considered robustly reachable – *all* values of uncontrolled input should lead to line 11. This restriction is *deliberate*. A more quantitative approach would hinder automation. For similar reasons, we limit ourselves to non-interactive scenarios, where the attacker input is chosen before uncontrolled input are known. We will further motivate these choices in Secs. 4.1 and 6.4.

Despite these deliberate restrictions, our case studies (Sec. 6.2) show the versatility of robust reachability. In the example above, we distinguish inputs

controlled by an attacker (a bad guy) from inputs which he cannot influence (see also *e.g.* libvncserver in Sec. 6.2). But with `doas` (Sec. 6.2), we distinguish inputs controlled by the system administrator (the good guy) from those which vary on each execution. Other situations are possible, for instance deterministic inputs versus non-deterministic ones like in the case of flaky tests [42] — where there are neither good nor bad guys. Robust reachability can help in all these situations either assessing the "quality" of a given trigger or test suite (criticality, replicability), generating "good" triggers or test suites, or proving their absence.

Table 2: Program constructs for which standard reachability yields fragile input

Randomisation based protections	Standard reachability models randomized or arbitrary values like canaries or ASLR as attacker-chosen values. This voids such protections. See also Fig. 1 and `libvncserver` in Sec. 6.2.
Uninitialized reads	With standard reachability, the attacker can choose the initial content of uninitialized memory. For example he can choose it to contain a password or a secret. See also `doas` in Sec. 6.2.
Underspecified initial state	A bug which is unreachable in normal operating conditions can become reachable if, *e.g.*, one leaves the stack location completely free. Then the bug only happens with pathological initial state.
Undefined behavior	A bug in a branch depending on undefined behavior is still *technically* reachable, but not robustly reachable. Note that even machine code has some undefined behaviors.
Interactions with the environment	Contrary to robust reachability, standard reachability lets the attacker use system calls and interactions by *e.g.* letting him choose the date to nanosecond precision, as if the environment helped him.
Opaque functions	One can abstract complex functions (crypto functions, `malloc`) as black boxes returning a fresh, symbolic value. Standard reachability allows the attacker to choose these values, yielding fragile triggers.

3 Background

Consider a program P and $\mathcal{S}$ the set of its possible states. Each state $s \in \mathcal{S}$ is labeled by a program location $\lambda(s) \in \mathcal{L}$. Execution of the program is represented by a (one-step) successor relation $\rightarrow \in \mathcal{S} \times \mathcal{S}$; its transitive reflexive closure is denoted by $\rightarrow^\star$. For a finite trace $t \in \mathcal{S}^\star$ and $s, s' \in \mathcal{S}$ two states, we write $s \rightarrow_t^\star s'$ if t starts with s, ends with s' and follows $\rightarrow$. The initial state $s_0(y)$ depends on the program input y. For a location $\ell \in \mathcal{L}$ and input y we write $y \vdash \ell$ if $s_0(y) \rightarrow^\star s$ where $\lambda(s) = \ell$. Additionally, for a trace $t \in \mathcal{S}^\star$, we write $y \vdash_t \ell$ if $s_0(y) \rightarrow_t^\star s$ where $\lambda(s) = \ell$. We use *trace* for successions of states and *path* for successions of locations. By abuse of notation, the path corresponding to a trace $t \in \mathcal{S}^\star$ is $\lambda(t) \in \mathcal{L}^\star$. For a path π, we denote its length $|\pi|$ and we write $y \vdash \pi$ if $\exists t \in S^\star.\, \lambda(t) = \pi \wedge y \vdash_t \ell$ where ℓ is the final location of π.

Definition 1 (standard reachability). *Given a program P, a location $\ell \in \mathcal{L}$ is reachable if $\exists y.\, y \vdash \ell$.*

It is often useful to consider the case of reaching a location ℓ with a state s satisfying some predicate ϕ. This can be reduced to standard reachability by adding `if (`ϕ`) /*new target*/` at the target location.

Definition 2 (correctness, completeness). *Let $\mathcal{V} : (P, l) \mapsto \{\mathbf{1}, \mathbf{0}\}$ be a verifier taking as input a program P and a location ℓ:*

- *$\mathcal{V}$ is correct when for all P, ℓ, if $\mathcal{V}(P, \ell) = \mathbf{1}$ then ℓ is reachable in P;*
- *$\mathcal{V}$ is complete when for all P, ℓ, if ℓ is reachable then $\mathcal{V}(P, \ell) = \mathbf{1}$;*
- *If $\mathcal{V}$ also takes an integer bound as input, $\mathcal{V}$ is k-complete when for all integers k and P, ℓ, if $\exists y.\, \exists t \in \mathcal{S}^\star.\, |t| \leq k \wedge y \vdash_t \ell$ then $\mathcal{V}(P, \ell, k) = \mathbf{1}$.*

In general, verifying reachability is undecidable, so verifiers cannot be both correct and complete. Correct verifiers can still be k-complete as k-completeness can be thought of as completeness for finite-path systems.

```
Data: bound k, target ℓ
for path π in GetPaths (k) do
    if π goes through ℓ then
        φ := GetPredicate(π)
        if ∃y. φ is satisfiable
          then
            return true
end
return false
```

(a) SE

```
Data: bound k, target ℓ
φ := ⊥
for path π in GetPaths (k) do
    if π goes through ℓ then
        φ := φ ∨ GetPredicate(π)
end
if ∃y. φ is satisfiable then return true

else return false
```

(b) BMC

Fig. 2: Reachability of ℓ with SE and BMC

Symbolic Execution (SE) and Bounded Model checking (BMC). SE [9] incrementally explores all paths in the program (up to, say, a bound k) and when an explored path reaches the target location ℓ, checks that this path is indeed executable. This is performed by converting a path π to an SMT formula pc_π, called *path constraint*, which has input y as its only free variable and is equivalent to $y \vdash \pi$, i.e., a path is executable if and only if its path constraint is satisfiable. Conversely, BMC [13] considers the program as a whole and builds a SMT formula expressing that one of the paths of length at most k leads to ℓ. It is equivalent to the disjunction of the path constraints of these paths. The target is reachable in k steps at most if and only if this formula is satisfiable.

These algorithms are detailed in Fig. 2, where `GetPredicate` turns a path into its path constraint and `GetPaths`(k) yields all paths below size bound k.

Proposition 1. *SE and BMC have the same expressive power: both are correct and k-complete.*

Interestingly, we show in Sec. 5 this is not true anymore with robust reachability.

Solvers. SE and BMC commonly discharge their satisfiability queries to SMT solvers [4] which take formulas as input, and output whether they are satisfiable

(along with a model) or not. Typical queries are expressed in the quantifier-free fragments of well known theories (linear integer arithmetic, bitvectors, arrays, *etc.*) where SMT solvers perform well in practice. In case of an undecidable theory, we can use incomplete solvers (possibly answering UNKNOWN), at the price of k-completeness.

4 Robust reachability

4.1 Definition

We introduce the new notion of *robust reachability*. We partition the input y into the *controlled input* a and the *uncontrolled input* x — we denote $y \triangleq (a, x)$. Let $\mathcal{A}$ and $\mathcal{X}$ be the sets of possible controlled and uncontrolled inputs respectively. A location is *robustly reachable* when the attacker can choose controlled input $a \in \mathcal{A}$ without having to rely on specific values of the uncontrolled input $x \in \mathcal{X}$ to reach his target. Input a is then called a *robust trigger* — otherwise it is a *fragile trigger*.

Definition 3 (Robust reachability). *A location $\ell \in \mathcal{L}$ is robustly reachable if $\exists a. \forall x. (a, x) \vdash \ell$. This definition depends on the partition of inputs.*

Proposition 2. *Robust reachability implies standard reachability. The converse implication does not hold.*

Discussion. As already mentioned at the end of Sec. 2, our definition of robust reachability specifically targets a threat model where the attacker speaks first, unaware of uncontrolled inputs. It deliberately excludes interactive systems where the attacker can choose some input, then receive some program output possibly leaking uncontrolled input, and then choose some more input *depending on what was received*. Modeling such situations requires additional quantifier alternations, which deeply impact the performance of proof methods and cripple automation, as shown in Sec. 6.4.

Likewise, a bug triggered for all uncontrolled inputs but one is not robustly reachable according to Definition 3. A quantitative definition of robust reachability could take into account the *proportion of uncontrolled inputs* triggering a bug. This hints at works about model counting [39,11], but the problem at hand is actually harder. Consider the following alternative definition: *(i)* find $a_{\max} \in \mathcal{A}$ such that a maximal proportion of uncontrolled inputs x lead to ℓ: $(a_{\max}, x) \vdash \ell$; *(ii)* measure how robustly ℓ can be reached by computing the proportion of uncontrolled inputs x such that $(a_{\max}, x) \vdash \ell$. Current model counting algorithms can only tackle problem *(ii)* along one path, and we argue in Sec. 6.4 that even *(ii)* alone is considerably more expensive than our SMT-based approach.

In other words, Definition 3 is a tradeoff to keep robust reachability amenable to automated verification. This does not prevent it from meeting its main goal: drawing the attention on more serious bugs. Some may of course be missed, but, as our case studies will show (Sec. 6), a good number will be found.

In the rest of this section, we review a few related properties and see how much they overlap with, but do not remove the need of, robust reachability.

4.2 Relation with non-interference

We partition inputs and outputs of a system into either *high* (highly classified) or *low* (public, e.g. observable). A system satisfies *non-interference* [31] when low outputs do not depend on high inputs, implying that secrets cannot leak. Robust reachability can be reformulated in a very non-interference-sounding phrasing: uncontrolled inputs (call them high) must not interfere with the attacker reaching the target location (the low output). Let us clarify this link.

Formally, let high input be uncontrolled input x, and low input be controlled input a. Let low output be whether control flow reached location ℓ. Non interference of the resulting system means that $\forall a, x, x'.\ ((a, x) \vdash \ell \iff (a, x') \vdash \ell)$.

Proposition 3. *If ℓ is (standardly) reachable and the system satisfies non-interference with the high/low partition described above, then ℓ is robustly reachable. The converse is false.*

Robust reachability requires a single value of the controlled input a for which reachability of ℓ is guaranteed but says nothing for other values of a, whereas non-interference constrains the system to behave much more independently of uncontrolled input than robust reachability but says nothing of reachability.

4.3 Interpretation in terms of hyperproperty

Robust reachability and its negation are not trace properties: the observation of a single trace is never enough to prove or disprove them. For example, observing a single trace reaching target ℓ with input (a, x) is both compatible with ℓ being robustly reachable (if all other inputs $(a, x'), x' \in \mathcal{X}$ also reach ℓ), and with ℓ not being robustly reachable (if some other x' is such that (a, x') does not reach ℓ). Robust reachability and its negation thus belong to the more general class of *hyperproperties* [16], i.e. statements relating several traces.

More specifically, Clarkson et al. [16] show that any hyperproperty is the intersection of a hypersafety hyperproperty (*i.e.* something bad cannot happen) and a hyperliveness hyperproperty (something good will eventually happen). Hypersafety is generally thought as easier to prove, notably with self-composition [6]. Unfortunately, robust reachability and its negation are pure hyperliveness in the general case: no finite set of finite traces can falsify them. However, in some conditions, they degenerate partly into hypersafety:

Proposition 4. *If the domain $\mathcal{X}$ of uncontrolled inputs is finite, then the negation of robust reachability is not pure hyperliveness (*i.e.*, it has a non-trivial hypersafety component).*

Proof. Robust reachability of ℓ can be proved by finding controlled input $a \in \mathcal{A}$ such that for all uncontrolled input $x \in \mathcal{X}$ one observes a trace starting with input (a, x) and reaching ℓ. When $\mathcal{X}$ is finite, this means that a finite observation can disprove non-(robust reachability). This is the definition of hypersafety.

This idea—trying to observe a hopefully small set of traces which together prove robust reachability—is crucial for algorithms and leads to our use of path merging in Sec. 5.3.

4.4 Interpretation in terms of temporal logic

Computational Tree Logic (CTL). CTL [14] is a temporal logic over the tree of possible traces. Let L be a labeling which maps states to the set of (atomic) predicates they satisfy. If ℓ is a predicate, the CTL formula ℓ is satisfied by all systems whose initial state s_0 verifies $\ell \in L(s_0)$. If ϕ is a CTL formula and s a state, then $\mathbf{EX}\phi$ expresses that ϕ holds in at least one (direct) successor of s, and $\mathbf{AF}\phi$ that all traces arising from s eventually reach a state from which ϕ holds. CTL introduces other operators, not needed here.

Proposition 5. *It is possible to express robust reachability with CTL.*

Proof. Let $\mathcal{S}' \triangleq \mathcal{S} \cup \mathcal{A} \cup \{s_i\}$ where s_i is a new state, let $\rightarrow' \triangleq \rightarrow \cup \{(s_i, a) \mid a \in \mathcal{A}\} \cup \{(a, s_0(a, x)) \mid a \in \mathcal{A}, x \in \mathcal{X}\}$, and let $L'(s)$ be equal to $L(s)$ if $s \in \mathcal{S}$ and $\varnothing$ otherwise. Then ℓ is robustly reachable if, and only if $\mathbf{EXAF}\ell$ is true in the new extended system $(\mathcal{S}', \rightarrow', L')$ with s_i as initial state.

HyperLTL. It is also possible to express robust reachability in the temporal logic HyperLTL[15], which allows to reason over sets of traces π, assuming we have an atomic predicate $\equiv_v$ stating that the first states of two traces have the same value for variable v. Robust reachability of ℓ can then be expressed as $\exists \pi. \forall \pi'. \mathbf{F}\ell_\pi \wedge (\pi \equiv_a \pi' \rightarrow \mathbf{F}\ell_{\pi'})$, where $\mathbf{F}\ell_\pi$ denotes that trace π goes through ℓ. In other words, there exists a trace π reaching ℓ s.t. all traces sharing the same controlled input also reach ℓ.

4.5 Robust reachability and automatic verification

The previous classification does not help us find an efficient *software verification method* for robust reachability. Indeed, while efficient CTL model checkers exists for the finite case [12] or very specific formalisms such as pushdown systems [47], most efforts in (general) software verification have been directed towards the verification of safety temporal formulas or simple termination [17] (formulas of the form $\mathbf{AF}\varphi$). Moreover, temporal logics like HyperLTL [15] suffer the same limitations, and checking for both reachability and non-interference is probably too strong a requirement in practice. Finally, one can prove the *absence* of robust reachability by proving the absence of standard reachability. It is thus possible to use existing algorithms for unreachability, based *e.g.* on invariant computation, at the price of even larger over-approximation than when they are used for their original purpose. This kind of approach is not our focus. In this paper we look for *correct verifiers* able to prove robust reachability (and report robust triggers) rather than to disprove it.

5 Automatically proving robust reachability

We now extend SE and BMC to the robust case.

5.1 Robust Bounded Model Checking

As mentioned in Sec. 3, BMC determines the reachability of a location ℓ by building a family of SMT formulas $\varphi_k(a, x)$ equivalent to $\exists t \in \mathcal{S}^\star. |t| \leq k \wedge (a, x) \vdash_t \ell$. φ_k expresses that ℓ is reachable in less that k steps. Then one proves that ℓ is reachable if and only if $\exists k. \exists a. \exists x. \varphi_k(a, x)$. This extends to robust reachability:

Proposition 6. *If the domain of uncontrolled input $\mathcal{X}$ is finite or the system has finitely many paths, then ℓ is robustly reachable if and only if $\exists k. \exists a. \forall x. \varphi_k(a, x)$.*

Proof. ($\Longleftarrow$) comes directly from the definition of φ_k. ($\Longrightarrow$). If ℓ is robustly reachable, let a_0 be a robust trigger. The set of paths P arising from inputs in $\{a_0\} \times \mathcal{X}$ is finite (bounded either by $\mathcal{X}$ or the number of paths in the system), and $\forall x. \bigvee_{\pi \in P} \mathrm{pc}_\pi(a_0, x)$ holds. Let $k = 1 + \max_{\pi \in P} |\pi|$. All paths in P are unrolled in φ_k so $\bigvee_{\pi \in P} \mathrm{pc}_\pi(a_0, x) \implies \varphi_k(a_0, x)$ and thus $\forall x. \varphi_k(a_0, x)$.

As a result, it is enough to replace the condition "$\exists y. \phi$ is satisfiable" by "$\exists a. \forall x. \phi$ is satisfiable" in Fig. 2b.

Corollary 1. *The resulting algorithm, robust BMC, is correct w.r.t. robust reachability. If the domain of uncontrolled input $\mathcal{X}$ is finite or the system has finitely many paths, then robust BMC is also k-complete.*

The finiteness hypothesis is required: if a program reaches a location after having executed a loop an unbounded, uncontrolled number of times, then robust BMC has to unroll an unbounded number of paths to prove robust reachability.

5.2 Robust Symbolic Execution

Similarly to BMC, we check that a path π robustly reaches the target by checking the satisfiability of $\exists a. \forall x. \mathrm{pc}_\pi(a, x)$, instead of $\exists a. \exists x. \mathrm{pc}_\pi(a, x)$. This means replacing "$\exists y. \phi$ is satisfiable" by "$\exists a. \forall x. \phi$ is satisfiable" in Fig. 2a. Unfortunately the resulting algorithm, robust SE, is not exactly what we want, as it proves a stronger property.

Definition 4 (Single-path robust reachability). *A location $\ell \in \mathcal{L}$ is single-path robustly reachable if $\exists \pi \in \mathcal{L}^\star. \exists a. \forall x. \exists t \in \mathcal{S}^\star. \lambda(t) = \pi \wedge (a, x) \vdash_t \ell$. In other words, the path used to reach ℓ is the same regardless of the uncontrolled input.*

Proposition 7. *Single-path robust reachability implies robust reachability. The converse implication does not hold.*

Proposition 8. *Robust SE is correct and k-complete w.r.t. single-path robust reachability.*

Proof. By construction, $\mathrm{pc}_\pi(a,x)$ is equivalent to $(a,x) \vdash \pi$ so $\exists\pi.\,\exists a.\,\forall x.\ \mathrm{pc}_\pi(a,x)$ is equivalent to single-path robust reachability of the last location of π.

Corollary 2. *Robust SE is correct but incomplete for robust reachability.*

Interestingly, the expressive powers of SE and BMC, which are the same for standard reachability, diverge when extended to robust reachability.

5.3 Path merging

Path merging [33] (a.k.a. state joining) consists in identifying "close" paths leading to the same location and replacing them by a *merged* path (summary). With original path constraints pc_{π_1} and pc_{π_2}, the merged path constraint is $\mathrm{pc}_{\pi_1} \vee \mathrm{pc}_{\pi_2}$. This is only an optimization in the standard setting, with no impact on k-completeness. The situation is different in the robust setting.

Data: bound k, target ℓ
1 $\phi := \bot$
2 **for** *path* π *in* `GetPaths` *(k)* **do**
3 **if** π *goes through* ℓ **then**
4 $\phi := \phi \vee$ `GetPredicate`(π)
5 **if** $\exists a.\,\forall x.\,\phi$ *is satisfiable* **then**
6 **return** true
7 **end**
8 **return** false

Algorithm 1: RSE+: Robust SE with systematic path merging

```
1  void main(a, x) {
2    if (x) x++; // π1
3    else x--;   // π2
4
5    if (!a) bug();
6  }
```

Fig. 3: An example where path merging is required

Consider the program in Fig. 3: the bug is robustly reachable with controlled input $a = 0$, but the control flow takes one of two paths π_1 and π_2 depending on the value x of uncontrolled input. This bug will not be found by robust SE as defined previously, as neither π_1 nor π_2 fulfills the satisfiability criterion $\exists a.\,\forall x.\ \mathrm{pc}_{\pi_i}(a,x)$. However, if π_1 and π_2 are merged, then the bug is found because $\exists a.\,\forall x.\ \mathrm{pc}_{\pi_1}(a,x) \vee \mathrm{pc}_{\pi_2}(a,x)$ is satisfiable. This leads us to robust SE with systematic path merging (RSE+, Alg. 1), better fit to robust reachability.

Proposition 9. *Robust SE with systematic path merging (RSE+) is correct for robust reachability. If the domain of uncontrolled input $\mathcal{X}$ is finite or the system has finitely many paths, then it is also k-complete.*

Proof. For k-completeness: If ℓ is robustly reachable, let a_0 be a robust trigger. The set of paths P arising from inputs in $\{a_0\} \times \mathcal{X}$ is finite (bounded either by $\mathcal{X}$ or the number of paths in the system). Let $k = 1 + \max_{\pi\in P} |\pi|$. For bound k, when `GetPaths` has output all paths in P, $\bigvee_{\pi\in P} \mathrm{pc}_\pi \implies \phi$ so $\exists a.\,\forall x.\,\phi$ is satisfiable.

In conclusion, *path merging improves the completeness of robust SE.* This is surprising because path merging is merely optional in standard SE.

5.4 Revisiting standard optimizations and constructs

Some optimizations commonly used in SE are not correct nor complete anymore in a robust setting. We show here how to adapt them.

Data: program entrypoint ℓ_0, bound k
1 $P := \{\ell_0\}$
2 **while** $P \neq \varnothing$ **do**
3 | Take a path π out of P
4 | **if** $|\pi| > k$ **then continue**
5 | **if** $\exists a, x.\ \mathrm{pc}_\pi$ unsat **then continue**
6 | **yield** π
7 | $P := P \cup \{\text{children paths of } \pi\}$
8 **end**

Algorithm 2: Implementation of `GetPaths` with path pruning

```
uncontrolled int x;
if (x<10)     { /* a */ }
else          { /* b */ }
/* c */
if (x>20) {
  /* d */
  if (x>30)   { /* e */ }
  else        { /* f */ }
}
```

Fig. 4: Failure case for universal path pruning

Data: entrypoint ℓ_0, bound k
$P := \{\ell_0\}$
while $P \neq \varnothing$ **do**
| Take a path π out of P
| **if** $|\pi| > k$ **then continue**
| **if** $\exists a. \forall x.\ \mathrm{pc}_\pi$ unsat **then**
| | `/* Skip MaybeMerge to disable path merging */`
| | $P :=$ `MaybeMerge`(π, P)
| | **continue**
| **end**
| **yield** π
| $P := P \cup \{\text{children paths of } \pi\}$
end

Algorithm 3: `GetPaths` with universal path pruning

1 **Function** `MaybeMerge`(π, P)
2 | Choose u a transitive child of the last location of π *(ideally, a strict postdominator of the second to last location of π)*
3 | Let π' the longest strict prefix of π.
4 | Let U the set of paths from π' to u
5 | **if** $\exists a. \forall x.\ \bigvee_{\pi'' \in U} \pi''$ is SAT **then**
6 | | Merge paths in U and add the result to P
7 | **end**
8 | **return** P

Algorithm 4: Incomplete path merging for universal path pruning

Incremental path pruning [48,3]. When a path has an unsatisfiable path constraint, all its descendent paths are also infeasible. For example, the path `acd` in Fig. 4 has path constraint $x < 10 \wedge x > 20$, which is unsatisfiable. One can prune this path, *i.e.* stop exploring it and its children `acde` and `acdf`.

In Fig. 2a this would be an optimization of `GetPaths`: as shown in Alg. 2, one checks that the path constraint of currently explored paths are satisfiable, and if not, the paths at fault are *pruned*, and their children paths are not explored. As a result, we now issue satisfiability queries in two occasions: during `GetPaths` to *prune* paths (Alg. 2, line 5), and when *validating* a candidate reaching path (Fig. 2a, line 5). Pruning queries and validation queries must be treated differently.

Robust SE is obtained from SE by adding a universal quantifier to *validation queries* but not *pruning queries*. The path constraint for path `a` in Fig. 4 is $\text{pc}_{\texttt{a}} = x < 10$ but $\exists a. \forall x.\ \text{pc}_{\texttt{a}}$ is false. Same applies for `b`. If we added a universal quantifier to pruning queries—which we call *universal path pruning*, see Alg. 3—we would prune `a` and `b`, and incorrectly conclude that `c` is not robustly reachable. In other words, Symbolic Execution with universal path pruning (denoted $\text{RSE}_\forall$) is correct but not complete.

Universal path pruning, however, conveys an interesting intuition: the full `if` branch below `acd` in Fig. 4 is not robustly reachable, because $\forall x. x > 20$ is false. With normal path pruning and RSE+, we would needlessly explore these paths. To take advantage of this, we keep $\text{RSE}_\forall$ but improve its completeness with path merging, as depicted in Alg. 4.

The main idea is that when a set of paths are to be pruned, they may pass the universal pruning test $\exists a. \forall x.$ pc when merged together. One way to find such sets of paths is the use the Control Flow Graph (CFG) of the program. For example when trying to prune $\pi = \texttt{a}$ in Fig. 4, we know by invariant of the set P of paths to be explored that $\pi' = \epsilon$ the empty path passes the universal test. We compute the strict postdominator $u = \texttt{c}$ of π': when the paths from π' to c join again, they pass the pruning test again. We then replace π by this merged path in the set P of paths to be explored.

Note that computing a postdominator is not required for correction. In our implementation, we cannot compute the exact CFG at the binary level so the chosen u may be wrong. In line 5 of Alg. 4 we check that we picked correctly, and otherwise, merging failed and we prune π. Despite the heuristic approach, the technique proves useful, as we will see in Sec. 6.

We denote Robust SE with universal path pruning and path merging as $\text{RSE}_\forall$+. It is correct and less incomplete than $\text{RSE}_\forall$.

Assumptions. It is common to model complex parts of the system by introducing their result as a symbolic input z and then *assume* that z satisfies the required properties. For example, Address Space Layout Randomisation (ASLR) for the stack pointer could be modeled by adding an *assumption* that $esp \in [m, M]$ where m and M are in-lined constant values. In standard SE this would be translated to an *assertion* $esp_0 \in [m, M]$ conjoined to

```
controlled unsigned int a;
uncontrolled unsigned int x;
assume(x < a);
if (false) bug();
```

Fig. 5: Unsound assumption, in pseudo-C.

the path constraint pc_π, where esp_0 is the initial value of esp. *Actually, in standard SE and BMC, assertions and assumptions are dealt with identically.*

In a robust setting, to the contrary, adding an *assumption* ψ to a path constraint yields $\psi \implies \mathrm{pc}_\pi$, while adding an *assertion* ϕ yields $\mathrm{pc}_\pi \wedge \phi$. Additionally, assumptions which mix controlled and uncontrolled inputs can make the algorithms above unsound without adaptation: in Fig. 5, reachability of `bug` maps to the SMT query $\exists a. \forall x. x < a \implies \bot$. It is satisfiable, with $a = 0$, which makes the premise false. However, this does not correspond to an executable path. Actually, formalizing robust reachability assuming $\psi(a, x)$ naively by $\exists a. \forall x. (\psi(a, x) \implies a, x \vdash \ell)$ does not imply standard reachability anymore. A slight adaptation is needed:

Definition 5 (Robust reachability under assumption). *A location ℓ is robustly reachable under the assumption of ψ when*

$$\exists a. ((\exists x. \psi(a, x)) \wedge (\forall x. (\psi(a, x) \implies (a, x) \vdash \ell)))$$

This definition preserves the implication from robust to standard reachability. The algorithms we presented are easily adapted to take it into account.

Interestingly, in the robust case, SE and BMC cannot handle assertions and assumptions in the same way anymore.

Concretisation and other optimizations. When path constraints along a path become too complex, some variables can be *concretized*: their symbolic value can be replaced by a concrete one [29,45,21]. Formally, concretizing a variable u to value 42 corresponds to adding an *assertion* $u = 42$. This sacrifices k-completeness for tractability. Actually, any additional constraint can be added, and several common optimizations (e.g., domain shrinking, path filtering) can be seen through this lens. These optimizations must be taken with care in the robust setting. First, considering them as assumptions instead of assertions would be incorrect. Second, if the value of the concretized variable ultimately depends semantically on uncontrolled input, the path does not pass universal validation anymore: for example, when concretizing x to 42, $\exists a. \forall x. \mathrm{pc}(a, x) \wedge x = 42$ is unsatisfiable because $\forall x. x = 42$ is false. As a result, locations visited further on this path become robustly unreachable. *In other words, concretisation only works on controlled or constant values.*

5.5 About constraint solving

Adaptations to robust reachability require solvers to deal with one alternation of quantifiers. Most theories become undecidable with quantifiers. Dedicated algorithms exist for a few decidable quantified theories, *e.g.* the array property fragment [7] or Presburger arithmetic [8]. For other theories, generic methods like E-matching [40] and MBQI [27] have proven rather efficient, although not complete. Sound approximations [25] also have been proposed to reduce quantified formulas to quantifier-free ones. In our experiments, the newly introduced quantifier associates to an increase in the frequency of time-outs and memory-outs, as seen in Sec. 6.3 and specifically Tab. 4.

6 Proof-of-concept of a robust symbolic execution engine

6.1 Implementation

We propose BINSEC/RSE, the *first* symbolic execution engine dedicated to robust reachability. We base our proof-of-concept on BINSEC [23], a binary executable formal analysis engine written in OCaml and already used in several significant case studies [20,19,43]. For the sake of experimental evaluation (Sec. 6.3) we actually implement five variants of robust reachability: **RSE** (basic approach in Sec. 5.2 with existential path pruning Sec. 5.4), **RSE+** (the same plus systematic path merging, Sec. 5.3), $\mathbf{RSE}_\forall$ (RSE with universal path pruning, Alg. 3), $\mathbf{RSE}_\forall$**+** (same, with path merging during path pruning, Alg. 4), and **RBMC** (Sec. 5.1). BINSEC/RSE emits quantified formulas in the theory of bitvectors and arrays (arrays are used to model memory) which are then solved by the quantified solver Z3 [22]. We reuse the recent ROW simplification [26] to reduces the number of array indexations. The source code of BINSEC/RSE, the test suite and the case studies of this section are available for reproduction at `https://github.com/binsec/cav2021-artifacts` and `https://zenodo.org/record/4721753`.

6.2 Case studies: exploitability assessment for vulnerabilities

We show here how BINSEC/RSE (unless otherwise specified, the RSE+ variant) can help in vulnerability assessment. Especially, we demonstrate that robust reachability allows deeper insights into a bug than standard reachability, by replaying 4 existing vulnerabilities.

CVE-2019-15900 in doas. `doas` is a utility granting higher privileges to users specified in a configuration file. User IDs are sometimes parsed incorrectly and left uninitialized. We look for a *vulnerable configuration file* denying root access to the attacker such that the (flawed) executable reliably grants root access to the attacker. For simplicity we assume that the system has no named users and groups and the configuration file has two lines.

BINSEC/RSE with standard reachability reports that root access is granted with a configuration file containing `permit :("@@@@@` when the initial memory address `0xffefffff` contains the group ID of the attacker and the stack starts at `0xfff0001f`. *This is a typical "false positive in practice": these conditions may vary unpredictably across executions so we cannot conclude regarding the exploitability of the flaw.*

With robust reachability where the configuration file is controlled but the initial state of memory is not, BINSEC/RSE reports in less than 10s that root access is granted reliably to the attacker when the configuration file contains `deny :4` and `permit b%@)@@(`. This is more useful, but `b%@)@@(` is not a valid user name. We test therefore if any other given user name is also affected by running the analysis with this user name concretized in the initial state. By this method, we proved that the flaw is also robustly reachable for `wwww`, a possible typo of a usual user name, as well as all two-letter lowercase user names.

In other words, if the system administrator grants privileges to a non existing user by mistake, he may unknowingly grant them to the attacker instead. *Here, robust reachability provides us with invaluable insight about the severity of a bug where standard reachability fails.*

CVE-2019-20839 in libvncserver. An attacker-chosen null-terminated string is copied by an unbounded `strcpy` into a 108-bytes buffer, leading to a stack buffer overflow. Exploitability is not guaranteed: null bytes cannot be copied, the executable is protected by SSP, *etc.* Starting from the vulnerable function, we ask whether it is possible to return to the address `0xdeadbeef`, chosen arbitrarily.

BINSEC/RSE reports that for standard reachability, the bug can be reached when: *(1)* the stack starts at `0xfff00000`; *(2)* the initial value of the return address of the function is 0; *(3)* the `gs` segment starts at `0xf7f00000`; *(4)* the stack canary is `0x01010180`; *(5)* neither system call in the function fails; *(6)* file descriptor 0 is free; *(7)* the input path has a specific value. *The attacker cannot prepare such a state, so this is another false positive in practice.*

With robust reachability, when only the input buffer is controlled and not the stack canary, BINSEC/RSE fails to prove or disprove exploitability in 24h. However, if we mark the canary as controlled, BINSEC/RSE finds an exploit in about 15 min. This suggests the canary brings a real protection against exploitation.

CVE-2019-14192 in U-boot. U-boot is an open-source boot-loader, popular for embedded boards. When booting over Network File System (NFS), U-boot does not validate the length field of some network packets. This length is subtracted 16 and used as a size to be copied. If a malicious packet declares a length of less than 16, computation underflows and leads to a buffer overflow.

We encode the situation as follows: the input network packet is controlled, the IP address of the victim is constant, the NFS state machine is initialized to expect the appropriate packet type and all other values are uncontrolled. BINSEC/RSE with the $\text{RSE}_\forall$+ variant (RSE+ times out here) proves in about 2 minutes that a memory copy of more than 4GB is robustly reachable, which is a strong indication of the criticality of this denial-of-service vulnerability.

CVE-2019-19307 in Mongoose. Mongoose is an embedded networking library. When receiving large MQTT packets, the length of the parsed packet can be computed as 0. The parsing loop does not advance and is thus infinite. We look for network packets whose length is parsed as 0 but are accepted as valid. BINSEC/RSE proves in less than a second that such situations are robustly reachable when only the network packet is controlled, confirming exploitability.

6.3 Experimental evaluation

Research Questions. We now seek to investigate in a more systematic way the following research questions:

RQ1 **Precision**: What is the best algorithm for robust reachability in terms of correctness and completeness?

Table 3: The 46 reachability problems selected for our evaluation

Type	Description		Controlled variable
Real	Vulnerability	CVE-2019-14192 (U-boot)	Network packet
		CVE-2019-20839 (libvncserver)	Socket path
		CVE-2019-19307 (mongoose)	Network packet
		CVE-2019-15900 (doas)	Configuration file
		CVE-2015-8370 (grub, simplified)	Password entry
	CTF	Flare-on 2015 1 & 2	Text entry
		Nintendo Coding Game	Input to hash function to invert
		Manticore	Text entry
	Function inversion	musl (strptime, strverscmp, atoi, strtol)	Preimage
		busybox (chmod mode and ip parsing)	
		μclibc (fnmatch)	
		openssl (base64 decoding)	
Synthetic	Motivating example of [25] and variants		Coefficients to affine function
	Motivating example of [24, Figure 2.2]		Text entry
	SSP bypass	See Sec. 2	Overflowing buffer
	ASLR bypass	2 examples	Various
	Undefined behavior	Overflow flag after 3-bit `shl` in `x86`	None
	Other	Various	Various

RQ2 **Gain associated to robustness**: Is standard SE subject to false positives and does robust reachability avoid them in practice?

RQ3 **Path pruning**: Does universal path pruning (Sec. 5.4) help explore less paths than normal path pruning?

RQ4 **Performance**: What is the overhead of robust reachability?

Protocol. We base our analysis on a set of 46 reachability problems on binary executables from various architectures (i686-windows-pc, i686-linux-gnu and armv7-linux-gnu) presented in Tab. 3. The average trace length for reachable problem instances is 809 instruction-long, with a maximum of 18k instructions. The problems fall into two categories: real code and synthetic examples (*e.g.* code designed to be analysed). For each executable, BINSEC/RSE determines if a certain location is robustly reachable from a certain initial state. If this is the case a model is output by BINSEC/RSE, and compared to a ground truth obtained by manual analysis. Tests were run on Intel Xeon E-2176M(12)@4.4GHz and we use Z3 4.8.7. Results are classified as follows:

Correct BINSEC/RSE proves the expected result, i.e. it either reports a robust trigger or rightfully proves the absence of such a trigger;

False positive a fragile trigger is reported;

Inconclusive BINSEC/RSE reports no trigger but search was incomplete or the solver returned UNKNOWN at some point;

Resource exhaustion timeout is an hour and memory usage is capped to 7GB.

Precision (RQ1). As expected, robust variants do not report any false positives, and path merging increases completeness. RSE variants with universal path pruning ($\text{RSE}_\forall$, $\text{RSE}_\forall+$) are less complete than those with existential path pruning, but they are less prone to timeouts. This is the case of CVE-2019-14192 in U-boot (Sec. 6.2), for example. RBMC suffers from path explosion (time out)

Table 4: Comparison of standard and robust algorithms over our 46 test cases

	SE	BMC	$\text{RSE}_\forall$	$\text{RSE}_\forall$+	RSE	RSE+	RBMC
Correct	30	22	30	34	37	44	32
False positive	16	14					
Inconclusive			16	11	7		1
Resource exhaustion		10		1	2	2	13
Total time (s)	2725	36911	3947	4374	13590	11534	47784
... w/o resource exhaustion	2725	911	3947	3589	6390	4334	984

much more often than RSE variants. *Overall, Robust SE with path merging and existential path pruning is the most promising method among those presented here, with 44/46 correct answers.* $\text{RSE}_\forall$*+ is less complete but terminates more often.*

Note that two interesting test cases in the "real" category of Tab. 3 need path merging to prove robust reachability: one where a pointer with uncontrolled alignment is passed to `memcpy`, and one where a branch depends on the result of IO. These situations are common programming idioms, demonstrating the importance of path merging.

Gain associated to robustness (RQ2). We compare standard SE with RSE+, the most precise algorithm of RQ1. *Standard reachability has about 30% false positives while robust reachability has none, at the cost of slightly more timeouts.*

There are no false positives in code in the "real" category, except in CVE replays. Our interpretation is that well-functioning programs are designed to behave the same regardless of the uncontrolled environment: concrete memory layout, stack canaries, *etc.* Robust reachability becomes decisive on buggy code, notably with undefined behavior. This is also illustrated by case studies (Sec. 6.2).

Path pruning (RQ3). We compare $\text{RSE}_\forall$, which features universal path pruning, to RSE, which features usual path pruning. Comparison is limited to test runs of more than a second which succeed with both methods. This is to prevent comparing a run where Binsec/RSE proves that the target is reachable and stops, to a run where Binsec/RSE does not find the target and explores the whole program. $\text{RSE}_\forall$ *explores 17% less paths and interprets 21% less instructions than RSE.* This comes at the price of more universally quantified SMT queries: the average time per SMT query goes up by 25%. Overall the run time of both methods is very close.

With path merging, the difference in paths explored disappears: $\text{RSE}_\forall$+ explores 1% less paths and instructions than RSE+. This is due to the fact that for some tests, path merging "unlocks" some new paths. Overall, $\text{RSE}_\forall$+ is 6% slower than RSE+ on successful, terminating tests.

Performance (RQ4). In this question, we compare the run time of robust algorithms to SE. Comparison is done on the same basis as before, except that we count timeouts. RSE+ is 74% slower than standard SE on geometric average.

This is mostly due to newly introduced time-outs (up to 260× slower) since median slowdown is only 15%. $\text{RSE}_\forall$ is more consistently slower with about 30% slowdown in both geomean and median. This is mainly explain by increased solver time (universal path pruning queries). $\text{RSE}_\forall$+ is close in median slowdown, but path merging introduces new timeouts and drives the average slowdown up to 62%. *RSE+ has a low overhead compared to standard SE, except for a few time-outs (2/46).*

6.4 Additional considerations

We excluded interactive systems and quantitative approaches from our definition of robustness (Definition 3, Sec. 4.1) to keep automated proof methods tractable. We motivate this choice by experimentally showing that these alternatives yield significant overhead. Technical details are provided in Appendix A.

Quantitative reasoning and model counting. We could imagine refining our definition of robust reachability, looking for some controlled input for which the number of uncontrolled inputs allowing to reach the intended target is maximal (or, above a certain threshold). Although we have already observed that model counters do not directly solve this problem (Sec. 4.1), we can lower bound its runtime cost by the cost of determining the number of uncontrolled x satisfying a path constraint for some given controlled input a_0. We experimentally measured it with SearchMC [39] and SMTApproxMC [11], two of the few model counters supporting the SMTlib2 format and the QF_BV theory. We compare this to our "all-or-nothing" qualitative approach on our 4 CVE case-studies: *the quantitative approach is here several orders of magnitude slower than our qualitative method* — SMTApproxMC always times out while SearchMC is at least 400× slower.

Interactive systems and quantifier alternations. We estimate the cost of adding more quantifier alternations in order to deal with interactive systems (Sec. 4.1), by modifying queries on the two of our case studies where interactive input makes sense (libvncserver and doas, *cf.* Sec. 6.2). *RSE+ in this setting does not terminate within 24h*, highlighting the fact that current SMT solvers have a very hard time generating models for quantified formulas beyond $\exists\forall$. It seems to be a fundamental issue as none of Z3 [22], Boolector [41] and CVC4 [5] is able to prove in less than 1h that $\forall z.\, \exists a.\, a \text{ XOR } 1 = z$ holds over 32-bit bitvectors.

7 Related work

Broadly speaking, we are interested in defining a subclass of *comparatively more interesting bugs* amenable to automation. We review related prior attempts.

Automatic exploit generation (AEG). These approaches seek to demonstrate the *impact* of a bug by automatically generating an exploit from it [1,10,36]. *This is complementary to robustness*, which focuses on replicability. Actually, both techniques could be advantageously combined, as a replicable exploit is

clearly more threatening than a fragile one. Current AEG methods being based on symbolic methods, adapting them for robustness looks feasible.

Quantitative reasoning & model counting. Several approaches rely on probabilities or counting to distinguish important issues from minor ones — for example (quantitative) *probabilistic model checking* [2,34] or *quantitative information flow analysis* [37]. Robust reachability could be refined in such a way. Yet, current quantitative approaches do not scale on software, as they often rely either on the finite-state hypothesis, or on *model counting* solvers [32], which are only at their beginning (see Secs. 4.1 and 6.4).

Flakiness. The opposition between *flaky* tests and *sturdy* tests [42, section 6.3] is close to that between robustly reachable bugs and normally reachable bugs. A test is flaky when it is reachable, but not robustly reachable under the partition of inputs where controlled inputs are deterministic inputs and uncontrolled inputs are non-deterministic inputs. *Flakiness is thus a particular case of (non-) robustness*. Especially, our tool can help find non-flaky tests.

Fairness. *Fairness assumptions* in model checking [35] aim at discarding traces considered as unrealistic and avoiding false alarms from the user point of view. While the goal is rather similar to ours, the two techniques are very different: fairness assumptions typically require certain sets of states to be visited infinitely often along a trace, while robust reachability requires that a trace cannot be influenced by uncontrolled input w.r.t. a given reachability property.

Symbolic Execution and quantifiers. Finally, while symbolic execution is commonly performed with quantifier-free constraints, a notable exception is *higher-order test generation* [28], where Godefroid proposes to rely on universally quantified uninterpreted functions ($\forall\exists$ queries) in order to soundly approximate opaque code constructs. Higher-order test generation and robust reachability are complementary as they serve two different purposes: robust reachability can only be used in a modest way for opaque code constructs (finding controlled inputs for which their value does not matter), while higher-order test generation is inadequate for robust reachability, as it would be as if the attacker could choose the controlled inputs knowing the uncontrolled ones.

8 Conclusion

We introduce the novel concept of robust reachability, that we argue is better suited than standard reachability in several important scenarios for both security (e.g., criticality assessment, bug prioritization) and software engineering (e.g., replicable test suites). We formally define and study robust reachability, discuss how standard symbolic methods to prove reachability can be revisited to deal with the robust case, design and implement the first robust symbolic execution engine and demonstrate its abilities in criticality assessment over 4 CVEs. We believe robust reachability is an important sweet spot in terms of expressiveness and tractability. We hope this first step will pave the way to more refinements and applications of robust reachability.

References

1. Avgerinos, T., Cha, S.K., Rebert, A., Schwartz, E.J., Woo, M., Brumley, D.: Automatic exploit generation. Communications of the ACM (2) (2014)
2. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Verifying continuous time Markov chains. In: CAV. Springer (1996)
3. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A Survey of Symbolic Execution Techniques. ACM Computing Surveys (3) (2018)
4. Barret, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability Modulo Theories. In: Handbook of Satisfiability. Ios press edn. (2009)
5. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. Springer (2011)
6. Barthe, G., D'Argenio, P., Rezk, T.: Secure information flow by self-composition. In: CSF'04 Workshop (2004)
7. Bradley, A.R., Manna, Z., Sipma, H.B.: What's Decidable About Arrays? In: VMCAI. Springer (2005)
8. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: Beyond Quantifier-Free Interpolation in Extensions of Presburger Arithmetic. In: VMCAI. Springer (2011)
9. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Communications of the ACM (2) (2013)
10. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing Mayhem on Binary Code. In: S&P 2012 (2012)
11. Chakraborty, S., Meel, K., Mistry, R., Vardi, M.: Approximate Probabilistic Inference via Word-Level Counting. AAAI **30**(1) (2016)
12. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: a new Symbolic Model Verifier. In: CAV'99. Springer (1999)
13. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: TACAS. Springer (2004)
14. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Logics of Programs. Springer (1982)
15. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal Logics for Hyperproperties. In: Principles of Security and Trust. Springer (2014)
16. Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security (6) (2010)
17. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond Safety. In: CAV. Springer (2006)
18. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks (1998)
19. Daniel, L.A., Bardin, S., Rezk, T.: Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In: S&P 2020. IEEE (2020)
20. David, R., Bardin, S., Ta, T.D., Mounier, L., Feist, J., Potet, M.L., Marion, J.Y.: BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis. In: SANER 2016. IEEE (2016)
21. David, R., Bardin, S., Feist, J., Mounier, L., Potet, M.L., Ta, T.D., Marion, J.Y.: Specification of concretization and symbolization policies in symbolic execution. In: ISSTA 2016. ACM (2016)
22. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS. Springer (2008)

23. Djoudi, A., Bardin, S.: BINSEC: Binary Code Analysis with Low-Level Regions. In: TACAS. Springer (2015)
24. Farinier, B.: Decision Procedures for Vulnerability Analysis. Ph.D. thesis, Université Grenoble-Alpes (2020)
25. Farinier, B., Bardin, S., Bonichon, R., Potet, M.L.: Model Generation for Quantified Formulas: A Taint-Based Approach. In: CAV. Springer (2018)
26. Farinier, B., David, R., Bardin, S., Lemerre, M.: Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing. In: LPAR-22 (2018)
27. Ge, Y., de Moura, L.: Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In: CAV. Springer (2009)
28. Godefroid, P.: Higher-order test generation. In: PLDI '11. ACM (2011)
29. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI 2005. ACM (2005)
30. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: Whitebox Fuzzing for Security Testing: SAGE has had a remarkable impact at Microsoft. Queue (1) (2012)
31. Goguen, J.A., Meseguer, J.: Security Policies and Security Models. In: S&P 1982. IEEE (1982)
32. Gomes, C.P., Sabharwal, A., Selman, B.: Model Counting. In: Handbook of Satisfiability. Ios press edn. (2008)
33. Hansen, T., Schachte, P., Søndergaard, H.: State Joining and Splitting for the Symbolic Execution of Binaries. In: Runtime Verification. Springer (2009)
34. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing (5) (1994)
35. Hart, S., Sharir, M., Pnueli, A.: Termination of Probabilistic Concurrent Program. ACM Transactions on Programming Languages and Systems (3) (1983)
36. Heelan, S.: Automatic generation of control flow hijacking exploits for software vulnerabilities. Master's thesis, University of Oxford (2009)
37. Heusser, J., Malacaria, P.: Quantifying information leaks in software. In: ACSAC '10. ACM Press (2010)
38. Holler, C., Herzig, K., Zeller, A.: Fuzzing with Code Fragments. In: 21st USENIX Security Symposium. USENIX Association (2012)
39. Kim, S., McCamant, S.: Bit-Vector Model Counting Using Statistical Estimation. In: TACAS. Springer (2018)
40. de Moura, L., Bjørner, N.: Efficient E-Matching for SMT Solvers. In: Automated Deduction – CADE-21. Springer (2007)
41. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0: System description. Journal on Satisfiability, Boolean Modeling and Computation **9**(1) (2015)
42. O'Hearn, P.W.: Incorrectness logic. POPL (2020)
43. Recoules, F., Bardin, S., Bonichon, R., Mounier, L., Potet, M.L.: Get Rid of Inline Assembly through Verification-Oriented Lifting. In: ASE 2019. IEEE (2019)
44. Reynolds, A., Tinelli, C., Goel, A., Krstić, S.: Finite Model Finding in SMT. In: CAV. Springer (2013)
45. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/FSE-13. ACM (2005)
46. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: SP 2016 (2016)
47. Song, F., Touili, T.: Efficient CTL model-checking for pushdown systems. Theoretical Computer Science (2014)
48. Williams, N., Marre, B., Mouy, P., Roger, M.: Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In: EDCC-05. Springer

A Details on the experiments supporting Sec. 6.4

We reuse the notations of the discussion in Sec. 4.1.

Model counting. For simplicity, consider single-path robust reachability of ℓ along a path with path constraint $\mathrm{pc}(a, x)$. It is equivalent to $\exists a. \forall x.\ \mathrm{pc}(a, x)$. A more quantitative approach would be to consider $a_{\max}$ s.t. the ratio $r(a_{\max})$ of x satisfying $\mathrm{pc}(a_{\max}, x)$ is maximal. The larger $r(a_{\max})$, the more robustly reachable ℓ. We try to experimentally get an idea of the cost of computing this. Determining $a_{\max}$ is an open problem, but we can lower bound the full computation time by the time to compute $r(a_{\max})$ from $a_{\max}$. As the algorithms below are randomized, we can measure the time to compute $r(a_0)$ for any a_0.

We collect the path constraint of the first path standardly reaching the target in our 4 case studies of Sec. 6.2. We arbitrarily choose a_0 satisfying $\exists x.\ \mathrm{pc}(a_0, x)$, and compare the time to (dis)prove $\forall x.\ \mathrm{pc}(a_0, x)$ with Z3 to the time to approximate $r(a_0)$ with two of the few model counters supporting SMTlib2 input in the QF_BV theory: SearchMC [39] (with tolerance $\varepsilon = 0.8$ and confidence $1 - \delta = 0.95$) and SMTApproxMC [11] (with tolerance $\varepsilon = 0.8$ and 1 iteration). We found no tool supporting arrays, so arrays were blasted. As shown in Tab. 5, the quantitative approach is orders of magnitude slower in all cases, and especially in the one case where it is indeed significantly more precise than our qualitative approach (u-boot).

Table 5: All-or-nothing (Z3) *vs* quantitative (SearchMC, SMTApproxMC) approaches: runtime and lower bound on $r(a_0)$. Timeout (TO) is $2,400$ seconds.

	doas		libvncserver		u-boot		mongoose	
Z3	0.02s,	0%	0.01s,	0%	0.07s,	0%	0.04s,	100%
SearchMC	9.4s,	10^{-13}	4.8s,	10^{-12}	190.6s,	25%	35.1s,	59%
SMTApproxMC	TO,	—	TO,	—	TO,	—	TO,	—

Quantifier alternations. We want to model a leak in ASLR in libvncserver (Sec. 6.2): the attacker knows about an address z and wants to use the bug to jump to z. The corresponding property is: for all values[4] of z, there exists an attacker input a such that for all other uncontrolled inputs x, control flow is diverted to z. This uses another universal quantifier, which we exclude in our definition of robust reachability to keep satisfiability queries tractable. We implemented this for libvncserver (additional quantification on the target jump address) and doas (additional quantification on the user and group ID of the attacker, and the typoed user name): RSE+ does not terminate within 24h.

[4] Without a null byte, but we ignore this detail for the sake of simplicity.