Formal Methods in System Design
https://doi.org/10.1007/s10703-022-00402-x

Introducing robust reachability

Guillaume Girol[1] · Benjamin Farinier[2] · Sébastien Bardin[1]

Received: 4 February 2022 / Accepted: 21 September 2022

Abstract
We introduce 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 5 existing vulnerabilities, and compare it with standard symbolic execution.

Keywords Symbolic execution · Reachability · Vulnerability assessment · SMT · Software verification · Program analysis

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) [10] or Bounded Model Checking (BMC) [14], we even have *in principle* the guarantee that the reported issue is real (*correctness*): there are no false positives.

✉ Guillaume Girol
guillaume.girol@cea.fr

✉ Sébastien Bardin
sebastien.bardin@cea.fr

Benjamin Farinier
benjamin.farinier@tuwien.ac.at

[1] CEA, List, Université Paris-Saclay, Gif-sur-Yvettes, France

[2] TU Wien, Vienna, Austria

Published online: 21 November 2022

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 [47]). For example, fuzzers are able to detect so many bugs [40] that they can lead to "bug triage issues" [32]. If each *replicable* (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 [39] or probabilistic reasoning [3, 36], insofar as they would hinder automation on real examples — these techniques are often either restricted to finite-state systems [3, 36] or rely on highly expensive model counting solvers [12, 41].

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.

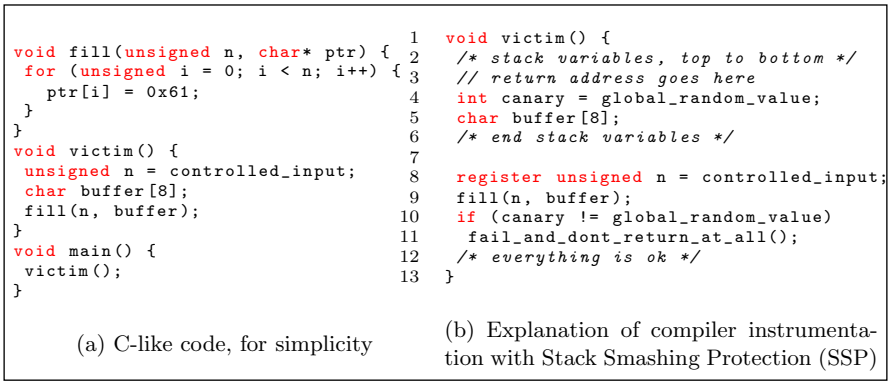
We typically focus on *security* scenarios where an *attacker* provides controlled input in one go, without knowledge of uncontrolled input — typically sending a malicious crafted file to obtain remote code execution or privilege escalation. We *deliberately* exclude interactive attack scenarios and 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 [15], hyperproperties [17] or hyper temporal logic [16], 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 SatisfiabilityModulo Theory (SMT) solvers [5] to generate models for *universally* quantified formulas [26, 28, 49], 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 (Sect. 4) along with the more general robust safety and guarantee property, and motivate its use (Sect. 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* (Sect. 4);
- We revisit Symbolic Execution (SE) [10] and Bounded Model Checking (BMC) [14] and show how they can be lifted to the robust case (Sect. 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



```
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

show how to adapt standard optimizations for Symbolic Execution and Bounded Model Checking;

– We implement and evaluate[1] (Sect. 6) the *first* symbolic execution engine dedicated to robust reachability, namely Binsec/RSE. We show how to use it for criticality assessment of 5 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.

Extension This work is a journal extension of a prior article published at CAV 2021 [29]. We have refined the text and added a total of 8 pages of new content, among which a new generalisation of the formal treatment from robust reachability of a location to a more general notion of robust reachability of any event (Sect. 4.1), a discussion on the link between robust reachability and the ATL temporal logic (Sect. 4.4), a discussion on lifting other properties and hyperproperties to their robust equivalent (Sect. 6.4.1), as well as some more detail on how we preprocess the universally quantified SMT formulas submitted to Z3 (Sect. 6.1). Finally, we also have added a new case study showing what robust reachability brings to the analysis of flaky tests (Sect. 6.2.2).

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.

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

[1] 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 [24]	Angr [51]	Robust reachability	BINSEC/RSE
No SSP	vulnerable	vulnerable ✓	vulnerable ✓	vulnerable ✓	vulnerable ✓	vulnerable ✓
SSP	protected	vulnerable ×	vulnerable ×	vulnerable ×	protected ✓	protected ✓

(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 like Stack Smashing Protection (SSP) [19]. 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.*

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 [51] or BINSEC [24] do report the bug as reachable (cf. Table 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 Table 2 several situations where standard reachability may lead to false positives, unlike robust reachability. Note that some cases (randomization 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

Table 2 Program constructs for which standard reachability yields fragile input

Randomization 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 Sect. 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 Sect. 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

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 Sections 4.1 and 6.4.

Despite these deliberate restrictions, our case studies (Sect. 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 Sect. 6.2). But with `doas` (Sectione 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 [47] — 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.

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 closure is denoted by $\rightarrow^+$. The set of finite traces is denoted by $\mathcal{S}^+$ and the set of infinite traces as $\mathcal{S}^\omega$. The set of all traces, finite or infinite, is $\mathcal{S}^\infty \triangleq \mathcal{S}^+ \cup \mathcal{S}^\omega$. For k an integer and t a trace, we denote by $t[k]$ its k-th state. The length of a trace t is $|t|$. We use *trace* for successions of states and *path* for successions of locations. By abuse of notation, the path corresponding to a trace $t \in \mathcal{S}^+$ is $\lambda(t) \in \mathcal{L}^+$. The initial state $t[1]$ is determined by a *program input* y chosen in some set $\mathcal{Y}$, yielding a mapping $s_1 \colon y \mapsto t[1]$. A program P is represented as the set of the traces that it can generate: $T(P) \subseteq \mathcal{S}^\infty$. $T(P)$ contains both maximal traces and their prefixes. We denote $t \preccurlyeq t'$ the fact that trace t is a prefix of trace t'.

Reachability is usually understood as control flow going through a location ℓ in the program. But we sometimes need to consider more complex properties like reaching a location of the program under some additional condition on the execution history. For example, in our motivating example of Sect. 2, we want to reach the return instruction together with an additional condition expressing that the return address was rewritten. Similarly, to prove the robust reachability of "use after free" bugs, we need to express a condition on the whole trace leading to the target. Specifically, we look for a trace in the set O of finite traces comprising three events in the right order: $p = \texttt{malloc}(\_)$, $\texttt{free}(p)$, and finally $*\,p$ for some value of p, such that p is not reallocated after being freed. For this reason we take a more general definition.

Definition 1 (Reachability) A set of finite traces $O \subseteq \mathcal{S}^+$ is reachable in program P if $T(P) \cap O \neq \emptyset$. We write $P \vdash R(O)$.

As a special case we get the usual property of reachability of a location:

Definition 2 (Reachability of a location) A location is reachable, denoted by $P \vdash R(\ell)$, if $P \vdash R(O)$ with $O = \{t \in \mathcal{S}^+ \mid \lambda(t[|t|]) = \ell\}$.

Finally, for a program $P \subseteq \mathcal{S}^\infty$, the restriction of this program to input y is defined by $T\big(P|_y\big) \triangleq \{t \in T(P) \mid t[1] = s_1(y)\}$, the restriction of P to path π by $T\big(P|^\pi\big) \triangleq \{t \in T(P) \mid \exists t' \in \mathcal{S}^+ t' \preccurlyeq t \wedge \lambda(t) = \pi\}$, and the restriction of P to bound $k \in \mathbb{N}$ by $T\big(P|^{\leq k}\big) \triangleq \{t \in T(P) \mid |t| \leq k\}$.

Definition 3 (correctness, completeness) Let $\mathcal{V} : (P, \Pi) \mapsto \{\mathbf{1}, \mathbf{0}\}$ be a verifier taking as input a program P and a property Π.

- $\mathcal{V}$ is correct with respect to Π when for all P, Π, if $\mathcal{V}(P, \ell) = \mathbf{1}$ then $P \vdash \Pi$;
- $\mathcal{V}$ is complete with respect to Π when for all P, Π, if $P \vdash \Pi$ then $\mathcal{V}(P, \ell) = \mathbf{1}$;
- If $\mathcal{V}$ is correct (resp. complete) for all $P|^{\leq k}$, then we say that it is k-correct (resp k-complete).

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.

Symbolic Execution (SE) and Bounded Model checking (BMC) Consider the problem of proving or disproving reachability of $O \subseteq \mathcal{S}^+$. The methods we are about to describe consist in encoding the possible witness traces in $O \cap T(P)$ as SMT formulas.

SE [10] incrementally explores all paths in the program (up to, say, a bound k). Each path π is converted into a SMT formula pc_π, called a *path constraint*, which expresses whether the program follows the path π for some input y (represented as a free variable). A solution y to pc_π is such that the trace t starting from y ($i.@e.@, t[1] = s_1(y)$) follows π: $\lambda(t) = \pi$.

Traditionnally, to prove reachability of a location ℓ, SE checks for each path π whether it contains ℓ and its path constraint pc_π is satisfiable. If there is such a path, then ℓ is reachable.

To handle the reachability of a general target $O \subseteq \mathcal{S}^+$, we consider *generalized path predicates* of the form $\text{pc}_\pi^O \triangleq \text{pc}_\pi \wedge \text{mon}_\pi^O$ where mon_π^O is a monitor expressing that the

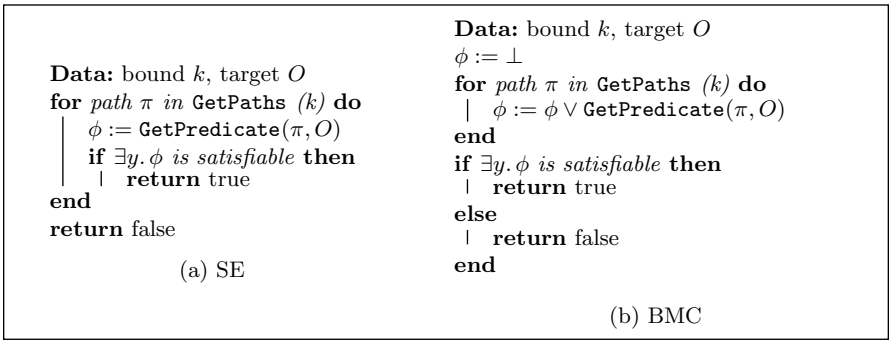


Fig. 2 Bounded proof attempt of $R(O)$ with SE and BMC

trace t corresponding to input y additionally reaches the target (there is a $t' \preccurlyeq t$ such that $t' \in O$).[2] In the end, SE iteratively checks satisfiability of pc_π^O for all enumerated paths π.

Conversely, BMC [14] considers the program as a whole (unrolled up to a bound k) and builds a SMT formula expressing that it contains a trace prefix in O. This formula is actually equivalent to the disjunction of the generalized path constraints of these paths.

These algorithms are given in Fig. 2, where $\texttt{GetPredicate}(\pi, O)$ turns a path into its generalized path constraint pc_π^O and $\texttt{GetPaths}(k)$ yields all paths below size bound k. Note that by abuse of notation, we will sometimes not precise that path constraints are generalized.

Proposition 1 *SE and BMC have the same expressive power*: *both are correct and k-complete with respect to reachability properties*.

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

Solvers SE and BMC commonly discharge their satisfiability queries to SMT solvers [5] 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

In this section we provide a formal definition of robust reachability, and argue why it deserves being singled out as a new problem rather than being viewed as a special case of a more generic framework like some of the more expressive temporal logics.

[2] We need to handle the case where the considered trace t is not in O, but one of its prefix is – which indeed satisfies reachability. Yet, in the cases where O is upward-closed ($i.@e.@$, closed by extension of trace) or $\texttt{GetPaths}(k)$ returns all path prefixes (and not only maximal paths), mon_π^O simply needs to express that $t \in O$.

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 4 (Robust reachability) A set of finite traces $O \in \mathcal{S}^+$ is robustly reachable in program P, denoted by $P \vdash \mathfrak{R}(O)$, if

$$\exists a \in \mathcal{A} \forall x \in \mathcal{X} P|_{(a,x)} \vdash R(O)$$

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

Discussion As already mentioned at the end of Sect. 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 Sect. 6.4.3.

Likewise, a bug triggered for all uncontrolled inputs but one is not robustly reachable according to Definition 4. 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 [12, 41], 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: $P|_{(a_{\max},x)} \vdash R(\ell)$; *(ii)* measure how robustly ℓ can be reached by computing the proportion of uncontrolled inputs x such that $P|_{(a_{\max},x)} \vdash R(\ell)$. Current model counting algorithms can only tackle problem *(ii)* along one path, and we argue in Sect. 6.4.3 that even *(ii)* alone is considerably more expensive than our SMT-based approach.

In other words, Definition 4 is a trade-off 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 (*Section 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* [33] 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 his goal (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 the target $O \subseteq \mathcal{S}^+$. Non-interference of the resulting system means that

$$\forall a, x, x' \left(P|_{(a,x)} \vdash R(O) \iff P|_{(a,x')} \vdash R(O) \right)$$

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

Robust reachability requires a single value of the controlled input a for which reachability of O 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.

Another way to put it is that robust reachability is looking for a value of controlled input a for which the system satisfies non-interference.

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). In such case one often resorts to the formalism of hyperproperties introduced by Clarkson and Emerson in [17].

A hyperproperty Π is represented as the set of programs P that satisfy it in the form of their set of traces: $\{T(P) \mid P \vdash \Pi\}$. Hyperproperties can relate several execution traces of a program. The price to pay is that observing a single trace is never enough to prove whether a hyperproperty is satisfied by the program. For hyperproperties, *observations* are defined as finite sets of finite, partial traces: we can observe the execution of the program on a finite set of inputs. This allows to define two important classes of hyperproperties: *hypersafety* for hyperproperties which can be disproved by an observation, and *hyperliveness* for hyperproperties where all possible (finite) observations are compatible with the hyperproperty being true. Informally, hypersafety expresses that something bad cannot happen, and hyperliveness that something good will always eventually happen.

Clarkson and Emerson [17] show that any hyperproperty is the intersection of a hypersafety hyperproperty and a hyperliveness hyperproperty. Hypersafety is generally thought as easier to prove, notably with self-composition [7]. 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 $O \subseteq \mathcal{S}^+$ (denoted by $\mathfrak{R}(O)$) can be proved by finding a controlled input $a \in \mathcal{A}$ such that for all uncontrolled inputs $x \in \mathcal{X}$, the trace starting with input

(a, x) belongs to O. When $\mathcal{X}$ is finite, this means that an observation (in the sense given above) can prove $\mathfrak{R}(O)$. In other words, an observation can disprove $\neg\mathfrak{R}(O)$, which is the definition of $\neg\mathfrak{R}(O)$ being 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 Sect. 5.3.

4.4 Interpretation in terms of temporal logic

We now show how robust reachability can be expressed by some sufficiently expressive temporal logics. Our definition of reachability of a set of traces O is so general that very few temporal logics can express it (for example CTL cannot express reachability of a bug where `malloc` is called strictly less often than `free` [43]). To make the effect of shifting from reachability to robust reachability more visible, we focus on the reachability of events expressible in the considered logics.

Computational Tree Logic (CTL) CTL [15] 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 $\mathbf{AF}\phi$ expresses that all traces arising from s eventually reach a state from which ϕ holds, $\mathbf{EX}\phi$ that ϕ holds in at least one direct successor of s, and $\mathbf{EF}\phi$ that ϕ holds in at least one transitive successor of s (this actually expresses reachability of ϕ). CTL introduces other operators, not needed here.

Proposition 5 *If ϕ is a CTL formula, then CTL can express robust reachability of ϕ.*

Proof Let $\mathcal{S}' \triangleq \mathcal{S} \cup \mathcal{A} \cup \{s_i\}$ where s_i is a new state, let $\rightarrow' \triangleq \rightarrow \cup \{(s_i, a) \mid a \in \mathcal{A}\} \cup \{(a, s_1(a, x)) \mid a \in \mathcal{A}, x \in \mathcal{X}\}$, and let $L'(s)$ be equal to $L(s)$ if $s \in \mathcal{S}$ and $\varnothing$ otherwise. Then ϕ is robustly reachable if, and only if $\mathbf{EXAF}\phi$ is true in the new extended system $(\mathcal{S}', \rightarrow', L')$ with s_i as initial state.

Alternating-Time Temporal Logic (ATL) ATL [1] is a temporal logic designed to model systems with multiple actors with distinct objectives. As usual the system is modeled by its set of states and its transition function, but each transition is decided by a set Σ of players: each player simultaneously makes a decision, and the actual transition is selected depending on these decisions. ATL formulas are the same as CTL, but operators $\mathbf{A}$ and $\mathbf{E}$ are generalized by a new operator $\langle\langle\cdot\rangle\rangle$. For a set of player $\Lambda \subseteq \Sigma$, $\langle\langle\Lambda\rangle\rangle\varphi$ means that there exists a strategy for players in Λ to make the system satisfy φ. At the limit, $\langle\langle\varnothing\rangle\rangle\varphi$ is $\mathbf{A}\varphi$ and $\langle\langle\Sigma\rangle\rangle$ means $\mathbf{E}\varphi$.

ATL contains CTL so Proposition 5 applies. However ATL makes it much more natural to express robust reachability since players can oppose each other. Consider $\Sigma = \{\epsilon, \alpha\}$ where ϵ is the environment, and α the attacker. Reachability of a formula ϕ is $\mathbf{EF}\phi$, or written otherwise: $\langle\langle\Sigma\rangle\rangle\mathbf{F}\phi$, which means that both the environment and attacker can cooperate to reach a state that satisfies ϕ. Robust reachability on the other hand is expressed by $\langle\langle\{\alpha\}\rangle\rangle\mathbf{F}\phi$, meaning that the attacker alone can make it so ϕ is satisfied. We can express the negation of robust reachability as well: $\langle\langle\{\epsilon\}\rangle\rangle\mathbf{G}\neg\phi$, which is a particular case of "collaborative invariance" [1].

HyperLTL It is also possible to express robust reachability in the temporal logic HyperLTL[16]. HyperLTL adds quantification over *trace variables*: $\exists\pi\phi$ means that there exists a trace starting in the initial state which satisfies ϕ and $\forall\pi\phi$ means that all traces

starting from the initial state satisfy ϕ. When using an atomic predicate ℓ we must specify over which trace variable we evaluate it: ℓ_π.

We consider a quantifier-free formula template ϕ where π is a free trace variable. Reachability of ϕ is $\exists \pi' \mathbf{F}\phi[\pi := \pi']$. For example, for reachability of ℓ, we take $\phi \triangleq \ell_\pi$. We assume 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 ϕ can then be expressed as $\exists \pi_1 \forall \pi_2 \mathbf{F}\phi[\pi := \pi_1] \wedge (\pi_1 \equiv_a \pi_2 \rightarrow \mathbf{F}\phi[\pi := \pi_2])$. In other words, there exists a trace π reaching ϕ such that all traces sharing the same controlled input also reach ϕ.

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 [13] or very specific formalisms such as push-down systems [52], most efforts in (general) software verification have been directed towards the verification of safety temporal formulas or simple termination [18] (formulas of the form $\mathbf{AF}\varphi$). HyperLTL [16] suffers the same limitations. As for ATL, it is so expressive (there can be arbitrarily many players, and arbitrarily many interactions between them and the system) that state of the art tools like STV [42] are limited to small models of a few dozens of states at best.

Moreover, checking for both reachability and non-interference as a correct, but incomplete proof method for robust reachability is probably too incomplete 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 discuss how to extend SE and BMC to the robust case.

5.1 Robust Bounded Model Checking

As mentioned in Sect. 3, BMC determines the reachability of $O \subseteq \mathcal{S}^+$ by building a family of SMT formulas $\varphi_k(a,x)$ equivalent to $P|^{\leq k} \vdash R(O)$. In the case of reachability of a location ℓ, φ_k expresses that ℓ is reachable in less that k steps. Then $R(O)$ holds 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 P has finitely many paths, then $P \vdash \mathfrak{R}(O)$ 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 W 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 $\forall x \bigvee_{\pi \in W} \mathrm{pc}_\pi^O(a_0, x)$ holds. Let

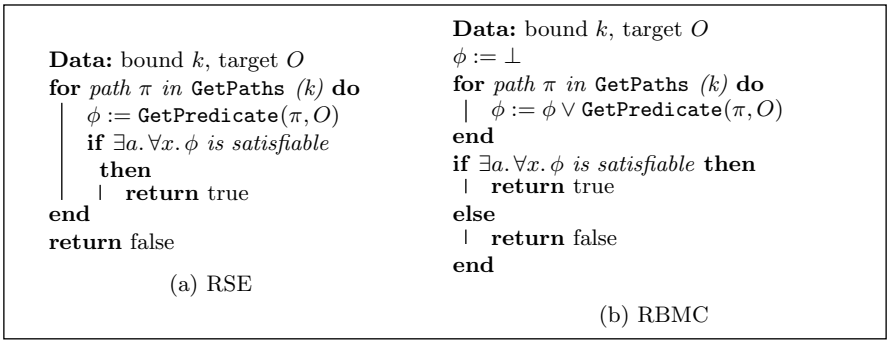


Fig. 3 Lifting SE and BMC to robust reachability

$k = 1 + \max_{\pi \in W} |\pi|$. All paths in P are unrolled in φ_k so $\bigvee_{\pi \in W} \mathrm{pc}_{\pi}^{O}(a_0, x) \implies \varphi_k(a_0, x)$ 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. The resulting algorithm, called Robust BMC, is presented in Fig. 3b. As solving universally quantified formulas is harder than the unquantified satisfiability checks required for standard BMC, we expect this new algorithm to trade performance for the finer precision that robust reachability provides.

Corollary 1 *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 $\exists a \forall x\, \mathrm{pc}_{\pi}^{O}(a, x)$, instead of $\exists a \exists x\, \mathrm{pc}_{\pi}^{O}(a, x)$. 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 (Fig. 3a), is not exactly what we want, as it proves a stronger property.

Definition 5 (Single-path robust reachability) A set O is single-path robustly reachable if $\exists \pi \in \mathcal{L}^{+} \exists a \forall x\, P|^{\pi}|_{(a,x)} \vdash R(O)$. In other words, the path used to reach O is the same regardless of the uncontrolled input.

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

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

Fig. 4 An example where path merging is required

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

Proof By construction, $\mathrm{pc}^{O}_{\pi}(a,x)$ is equivalent to $P|^{\pi}|_{(a,x)} \vdash R(\ell)$, therefore $\exists\pi\exists a\forall x\,\mathrm{pc}^{O}_{\pi}(a,x)$ 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 [35] (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 $\mathrm{pc}^{O}_{\pi_1}$ and $\mathrm{pc}^{O}_{\pi_2}$, the merged path constraint is $\mathrm{pc}^{O}_{\pi_1} \vee \mathrm{pc}^{O}_{\pi_2}$. 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 O
1  φ := ⊥
2  for path π in GetPaths (k) do
3  |  φ := φ ∨ GetPredicate(π, O)
4  |  if ∃a.∀x.φ is satisfiable then
5  |  |  return true
6  end
7  return false
```

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

Consider the program in Fig. 4: 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 $\exists a\forall x\,\mathrm{pc}^{O}_{\pi_i}(a,x)$. However, if π_1 and π_2 are merged, then the bug is found because $\exists a\forall x\,\mathrm{pc}^{O}_{\pi_1}(a,x) \vee \mathrm{pc}^{O}_{\pi_2}(a,x)$ is satisfiable. This leads

us to robust SE with systematic path merging (RSE+, Figure 1), better fit to robust reachability.

Proposition 9 *Robust SE with systematic path merging (RSE+) is correct for all robust reachability properties. 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 $\mathfrak{R}(O)$ holds, 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, $\bigvee_{\pi \in P} \mathrm{pc}_\pi^O \implies \phi$ 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 entry point ℓ_0, bound k
1 $P := \{\ell_0\}$
2 **while** $P \neq \varnothing$ **do**
3 Take a path π out of P
  `/* If too long, discard` π `*/`
4 **if** $|\pi| > k$ **then continue**
  `/*` pc_π `expresses that the path is executable */`
5 **if** $\exists a, x.\ \mathrm{pc}_\pi$ unsat **then continue**
6 **yield** π `// return` π `to caller`
7 $P := P \cup \{\text{children paths of } \pi\}$
8 **end**

Algorithm 2: Implementation of `GetPaths` with path pruning

Fig. 5 Failure case for universal path pruning

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

```
Data: entrypoint ℓ0, bound k
P := {ℓ0}
while P ≠ ∅ do
    Take a path π out of P
    if |π| > k then continue
    if ∃a.∀x. pc_π unsat then
        /* Skip MaybeMerge to
           disable path
           merging                */
        P := MaybeMerge(π, P)
        continue
    end
    yield π
    P := P∪{children paths of π}
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 ∃a.∀x. ⋁_{π″∈U} π″ 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 [4, 55] The (non-generalized) path constraint pc_π expresses that a path π is executable. We can use this to perform an optimization called *incremental path pruning*. When a (partial) path has an unsatisfiable path constraint pc_π, all its descendant paths are also infeasible. For example, the path `acd` in Fig. 5 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 Algorithm 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 (Algorithm 2, line 5), and when *validating* a candidate reaching path (Fig. 2a, line 4). 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. 5 is $\text{pc}_{\texttt{a}} = x < 10$ but $\exists a \forall x\, \text{pc}_{\texttt{a}}$

Fig. 6 Unsound assumption, in pseudo-C

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

is false. Same applies for `b`. If we added a universal quantifier to pruning queries—which we call *universal path pruning*, see Algorithm 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 by $\text{RSE}_\forall$) is correct but not complete.

Universal path pruning, however, conveys an interesting intuition: the full `if` branch below `acd` in Fig. 5 is not robustly reachable, because $\forall xx > 20$ 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 Algorithm 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 to use the Control Flow Graph (CFG) of the program. For example when trying to prune $\pi = \texttt{a}$ in Fig. 5, we know by invariant of the set P of paths to be explored that the empty path $\pi' = \epsilon$ 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 Algorithm 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 Sect. 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 Randomization (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 the path constraint pc, 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 $\psi \implies \text{pc}$, while adding an *assertion* ϕ yields $\text{pc} \wedge \phi$. Additionally, assumptions which mix controlled and uncontrolled inputs can make the algorithms above unsound without adaptation: in Fig. 6, reachability of `bug` maps to the SMT query $\exists a \forall xx < a \implies \bot$. 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 $\exists a \forall x(\psi(a, x) \implies P|_{(a,x)} \vdash R(\ell))$ does not imply standard reachability anymore. A slight adaptation is needed:

Definition 6 (Robust reachability under assumption) O is robustly reachable *under the assumption of* ψ if

$$\exists a((\exists x \psi(a, x)) \wedge (\forall x(\psi(a, x) \implies P|_{a,x} \vdash R(O))))$$

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 [22, 31, 50]. 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 [8] or Presburger arithmetic [9]. For other theories, generic methods like E-matching [45] and MBQI [28] have proven rather efficient, although not complete. Sound approximations [26] 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 Sect. 6.3 and specifically Table 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 [24], a binary executable formal analysis engine written in OCaml and already used in several significant case studies [20, 21, 48]. For the sake of experimental evaluation (Sect. 6.3) we actually implement five variants of robust reachability: **RSE** (basic approach in Sect. 5.2 with existential path pruning Sect. 5.4), **RSE+** (the same plus systematic path merging, Sect. 5.3), $\mathbf{RSE}_\forall$ (RSE with universal path pruning, Algorithm 3), $\mathbf{RSE}_\forall$**+** (same, with path merging during path pruning, Algorithm 4), and **RBMC** (Sect. 5.1).

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.

Solving universally quantified SMT formulas Binsec/RSE emits quantified formulas in the theory of bitvectors and arrays (arrays are used to model memory) which are then solved by the solver Z3 [23]. Z3 supports universally quantified formulas quite well, but falls short when arrays are quantified. This is a problem as initial memory is an array, and

in most threat models, should be labeled as uncontrolled. As an example, Z3 is actually not able to prove the unsatisfiability of a formula as simple as

$$\exists a \forall mem.\, mem[42] = a \tag{1}$$

First we reuse the recent ROW simplification [27] to reduces the number of array indexations. In favorable cases, this simplification alone can even simplify all arrays out.

To deal with cases like (1) that remain after ROW simplification, we implemented one further simplification that moves the memory out of the universal quantifier. As the ROW simplification is quite powerful, it is only needed infrequently, and when it does, it is hard to reason about the root causes of the failure due to the complexity of the resulting formulas, but for the sake of illustration, let us modify one simpler test case to exhibit this behavior. Consider our function inversion test relating to musl's implementation of `strtol`. It looks for a controlled string s such that $\texttt{strtol}(s) = 42$. If we do not initialize the memory of an internal lookup table, then $\text{RSE}_\forall$ fails on a formula corresponding to

$$\exists s \forall mem.\, mem[s[3] + table] \geq 10$$

where *table* is the offset of the table in the executable. Z3 returns UNKNOWN on this formula.

Let us now explain the transformation informally. In the case of eq. 1, we would like to rewrite it into $\exists a, mem \forall v(store(mem, 42, v))[42] = a$. This corresponds to initialising all memory locations that are later read with an uncontrolled, fresh value. As we are sure the original memory *mem* is never read, we can move it out of the universal quantifier. In our experience, Z3 deals easily with formulas where only bitvectors, as opposed to arrays, are universally quantified. The general case can be more involved, as the locations where memory is read can be symbolic, and even depend on memory. We therefore introduce one more layer of indirection:

$$\exists a \forall mem.\, mem[mem[12] + 1] = a$$

would be transformed to

$$\exists a, mem\, \forall v, i.\, \text{let}\, mem' = store(mem, i, v) \text{in}\, i = mem'[12] + 1 \implies mem'[42] = a$$

When dealing with n reads inside the original formula, we need to introduce n symbolic indices, and deal with all possible equalities between them, so the transformation actually yields a formula of size $O(n^2)$. This can be problematic, but already said, this is mostly a fallback when ROW does not simplify arrays out already. We thus expect very few memory reads to remain.

6.2 Case studies

6.2.1 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 5 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.

```
static int parseuid(const char *s, uid_t *uid) {
    const char *errstr;
    sscanf(s, "%d", uid);
    if (errstr) return -1;
    return 0;
}
```

This code is used to parse user IDs allowed to execute commands. If this function erroneously returns the attacker's user ID in the parameter `uid`, then privileged escalation is possible. When `s` is not the text representation of an integer, `uid` remains uninitialized memory. The branch `if (errstr)` was optimized out when we compiled. The exact same flaw is present in the function parsing group IDs.

Fig. 7 Code responsible for CVE-2019-15900

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 user and group, the configuration file has two lines and the attacker has uid 4 and gid 7.

BINSEC/RSE with standard reachability reports that root access is granted 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 10 s that root access is granted reliably to the attacker when the configuration file contains `deny:4` and `permit b%@)@@(`. When parsing the first rule, `parseuid` correctly initialises the `uid` variable of Fig. 7 to the uid of the attacker, 4.

The next rules allows root access to any non-existing username, `parseuid` leaves this variable untouched and privileged escalation is possible.

This result is considerably 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 24 h. 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 min 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.

CVE-2015-8370 in Grub (aka back to 28) Grub is a boot-loader used in most Linux systems. The original vulnerability is an integer underflow leading to buffer underflow when the user types 28 times on backspace on the password prompt of grub. We extracted the vulnerable function, ported it to Linux and simplified it so that it overwrites a local variable instead of the Interrupt Vector Table which is not easily modelled with BINSEC. BINSEC/RSE proves in 17 s that the vulnerability is robustly reachable.

6.2.2 Flaky tests

Consider the test suite of a program. Ideally, it should fail when the program is incorrect, and succeed when no buggy code path has been exercised. A test is flaky when its outcome is non-deterministic. This is undesirable and recent work [47] looked into reasoning about such tests.

Robust reachability can be used not only to detect flaky tests, but also to choose the inputs to pass to the function to be tested to make the test non flaky.

Detection. Flakiness can be seen as a special case of non-robustness when labeling non-deterministic inputs as uncontrolled: a test is flaky when the "success" outcome is not robustly reachable.

Actually, the full expressivity of robust reachability is not necessary to characterize flaky tests, as a test normally has no explicit input, only implicit, uncontrolled inputs. Therefore the property of interest for a whole test is "for all implicit inputs, success is reached".

Sturdy input generation. Consider Fig. 8 where we test the functionality of function `foo`. This test is flaky, as line 7 is not robustly reachable. We want to fix the flakiness of this test by finding a value for input `x` of function `foo` that makes the test deterministic. We mark `x` as a controlled, symbolic input and leave `nondet` as uncontrolled input.

Success becomes robustly reachable, and BINSEC/RSE even reports that `x = 2` guarantees deterministic test execution. This allows us to fix our test, and this time, we really used the full power of robust reachability.

Additionally, consider the CFG of function `foo`: line 4 is robustly reachable but not its children lines 5 and 6. This is the sign that it is actually a *source of flakiness*. And indeed, we can modify the conditional at line 4 so that all lines in `foo` become robustly reachable.

6.3 Experimental evaluation

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

1. **Precision**: What is the best algorithm for robust reachability in terms of correctness and completeness?
2. **Gain associated to robustness**: Is standard SE subject to false positives and does robust reachability avoid them in practice?
3. **Path pruning**: Does universal path pruning (Sect. 5.4) help explore less paths than normal path pruning?
4. **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 Table 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 analyzed). 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-2176 M(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, see for example CVE-2019-14192 in U-boot (Sect. 6.2). RBMC suffers from path explosion (time out) 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.*

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 [26] and variants		Coefficients to affine function
	Motivating example of [25, Figure 2.2]		Text entry
	SSP bypass	See Sect. 2	Overflowing buffer
	ASLR bypass	2 examples	Various
	Undefined behavior	Overflow flag after 3-bit `shl` in `x86`	None
	Other	Various	Various

```
1  void foo (int x) {
2  ●  if (x % 2 == 0) {
3  ●      return;
4  ●  } else if (!nondet) {
5  ○      error();
6     }
7  ○  return;
8  }
9  int main() {
10    int x = 3;
11    foo(x);
12    x += 2;
13    if (x!=4) { error(); }
14    return 0;
15 }
```

In function `foo`, robustly reachable nodes when `x` is symbolic are marked as ● and non robustly reachable as ○.

Fig. 8 Example of flaky test adapted from [47]

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

	SE	BMC	$RSE_{\forall}$	$RSE_{\forall}+$	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

RSE: Robust Symbolic Execution. RBMC: Robust Bounded Model Checking. + in acronyms denotes path merging, and ∀ universal path pruning

Note that two interesting test cases in the "real" category of Table 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 (Sect. 6.2).

Path pruning (RQ3) We compare $RSE_{\forall}$, 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. *$RSE_\forall$ 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: $RSE_\forall$+ 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, $RSE_\forall$+ 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%. $RSE_\forall$ 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). $RSE_\forall$+ 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

6.4.1 Going beyond reachability

The formal framework we provide in Sect. 4.1 allows to lift any reachability property to its robust equivalent. Could we do the same for other classes or properties and hyperproperties? Formally, yes, but we may need to give up part of the results of this paper, and in the case of hyperproperties, the lifted property might even be nonsensical, so some care must be taken.

Robust trace properties A trace property is a set of traces $\Pi \subseteq \mathcal{S}^\infty$. A program P satisfies Π if $P \subseteq \Pi$. One can apply the same construction as in Definition 4 to obtain a lifted "robust" property $\mathfrak{R}(\Pi)$: P satisfies $\mathfrak{R}(\Pi)$ if $\exists a \forall x P|_{(a,x)} \vdash \Pi$.

For example, consider non-termination. Robust non-termination expresses that for some controlled input, the program is guaranteed not to terminate. In a security context, this encodes a form of "guaranteed denial-of-service".

A well studied class of trace properties is the class of *safety properties*. They are the negations of the reachability properties as defined in Definition 1, or described more intuitively, they are trace properties that can be falsified by observing a bad finite trace prefix. As an example of safety property, consider the absence of null pointer dereference. Lifting it like before, "robust absence of null pointer dereference" expresses that for some controlled input, the program is guaranteed to be free of null pointer dereference. This property is weaker that the original one, and makes little sense with the threat model we used until now: why would we rely on the attacker to establish safety of our program? We actually need to reverse this threat model: consider the controlled input as controlled by the system administrator trying to harden the system. Then the property consists in looking for a system configuration which is impervious to attacks.

While the construction we introduced still works for trace properties, the proof methods of Sect. 5 do not. Developing algorithms to prove robust trace properties is left to future work.

Robust hyperproperties Informally, robustness lifts a trace property Π by adding quantification over the inputs of the system, therefore a quantification over traces: there must be a controlled input, such that all traces starting with this input satisfy Π. But since hyperproperties (introduced in Sect. 4.3) are also allowed to quantify over traces, the quantifications might collide. Consider the hyperproperty Π "the program terminates on average in less than 100 steps". Lifting Π to robustness would yield "there exists a controlled input, such that for all uncontrolled inputs, the only trace starting with those inputs terminates in average in less than 100 steps." Obviously, "average" here lost its meaning.

We can solve this issue by generalizing the construction of Definition 4. We now split inputs to the program into three parts: controlled inputs a, uncontrolled input x, and remaining input y. The lifted hyperproperty only quantifies on y. In the example above, Q becomes "there exists a such that for all x, the system terminates in less than 100 step on average on y".

Formally, we need to adapt the restriction of a program P to only partial inputs: $P|_{(a,x,\cdot)} = \{t \in P \mid \exists y[1] = s_1(a,x,y)\}$. Then, lifting the hyperproperty Π yields:

$$\{P \subseteq \mathcal{S}^\infty \mid \exists a.\forall x.\, P|_{(a,x,\cdot)} \vdash \Pi\}$$

Whether this construct has a useful meaning is very context-dependent. But let us give an example for non-interference, or rather its negation. Consider the case where the attacker is only one of many unprivileged users, and has the goal of breaking non-interference, i.e. observing low outputs that leak information on the inputs of privileged users. We label as controlled the inputs a of the attacker, and denote by x and y the inputs of other unprivileged and privileged users respectively. If $(a,x,y) \sim (a',x',y')$ denote that traces starting with inputs (a, x, y) and (a',x',y') are observationally equivalent (be it termination-sensitive or not, for the sake of simplicity), then one would write non-interference as:

$$\forall a, x, y,'\, y(a,x,y) \sim (a,x,y')$$

Robust violation of non-interference is then

$$\exists a.\forall x.\exists y, y'(a,x,y) \nsim (a,x,y')$$

which means that the attacker can choose wisely a controlled input a such that, whatever other unprivileged users do, a leak of information on high input y is possible. Here we only let non-interference quantify over y.

6.4.2 Negation of robust reachability

We focus in this paper on (positively) proving robust reachability and discussing the potential applications for security assessment. Let us briefly discuss now the case of proving the *negation* of robust reachability. This property, which falls in the category of *collaborative invariance* [1], expresses that for all controlled inputs a, there exists an uncontrolled input x that prevents some event O. In other words, it is always possible (for the system, for the defender, *etc.*@) to preserve the invariant $\neg O$. While this is weaker than $\neg O$, it is still relevant for security as it characterizes those systems that may not be fully secure (the invariant does not hold) but which can still always be defended. From a broader perspective, this can

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		grub	
Z3	0.02 s	0%	0.01 s	0%	0.07 s	0%	0.04 s	100%	0.07 s	100%
SearchMC	9.4 s	10^{-13}	4.8 s	10^{-12}	190.6 s	25%	35.1 s	59%	0.7 s	66%
SMTApproxMC	TO	–	TO	–	TO	–	TO	–	186 s	7%

be an interesting step toward a principled definition of soundiness [44]: instead of discarding a system because it does not uphold the expected invariant, we can still show that it can be made to work, and thus see more finely the value we can attach to it.

6.4.3 Scope of the definition

We excluded interactive systems and quantitative approaches from our definition of robustness (Definition 4, Sect. 4.1) to keep automated proof methods tractable. We motivate this choice by experimentally showing that these alternatives yield significant overhead.

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

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 (Sect. 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.

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 $a_{\max}$ such that 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 5 case studies of Sect. 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 [41] (with tolerance $\varepsilon = 0.8$ and confidence $1 - \delta = 0.95$) and SMTApproxMC [12] (with tolerance $\varepsilon = 0.8$ and 1 iteration). We found no tool supporting arrays, so arrays were blasted. As shown in Table 5, *the quantitative approach is at least an order of magnitude slower than our qualitative method* — SMTApproxMC mostly times out while SearchMC is 350× slower in geometric mean. Ironically, the overhead is highest in the one case (u-boot) where the quantitative approach is actually significantly more precise than our qualitative approach.

Quantifier alternations Assume we want to model a leak in ASLR in libvncserver (Sect. 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[3] of z, there exists an attacker input a such that

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

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 (Sect. 4.1) to keep satisfiability queries tractable. Similarly, in our case study on doas, we would like to check that the exploit works for any typoed username and, and for any user ID and group ID.

We implemented this and in both cases, RSE+ does not terminate within 24 h. This is not a scaling issue but a more fundamental one with additional quantifier alternations: none of Z3[23], Boolector[46] and CVC4[6] are able to prove in less than 1 h that $\forall z \exists aa \text{ XOR } 1 = z$ holds, with 32-bit bitvectors (where the quantification of x is even omitted).

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 [2, 11, 38]. *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* [3, 36] or *quantitative information flow analysis* [39]. 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 [34], which are only at their beginning (see Sections 4.1 and 6.4).

Fairness Fairness assumptions in model checking [37] 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* [30], 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.

Alternative proof methods Some recent methods based on abstract interpretation [53, 54] may be a good starting point to prove robust reachability. Notably, FuncTion [53] can infer preconditions over inputs for guarantee properties. If some inferred precondition is not empty and does not depend on x, then robust reachability holds. Interestingly this technique has been generalized to arbitrary CTL formulas [54], in which robust reachability can be encoded. Yet, whether this method can be extended beyond integer-manipulating program while remaining precise enough is still unclear.

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 5 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.

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

References

1. Alur R, Henzinger TA, Kupferman O (2002) Alternating-time temporal logic. J ACM 49(5):672–713
2. Avgerinos T, Cha SK, Rebert A, Schwartz EJ, Woo M, Brumley D (2014) Automatic exploit generation. Communicat ACM 57(2):74–84
3. Aziz A, Sanwal K, Singhal V, Brayton R (1996) Verifying continuous time Markov chains. In: CAV. Springer
4. Baldoni R, Coppa E, D'elia DC, Demetrescu C, Finocchi I (2018) A Survey of Symbolic Execution Techniques. ACM Comput Survey 51(3):1–39
5. Barret CW, Sebastiani R, Seshia SA, Tinelli C (2009) Satisfiability Modulo Theories. In: Handbook of Satisfiability. Ios press edn
6. Barrett C, Conway CL, Deters M, Hadarean L, Jovanović D, King T, Reynolds A, Tinelli C (2011) CVC4. In: CAV. Springer
7. Barthe G, D'Argenio P, Rezk T (2004) Secure information flow by self-composition. In CSF'04 Workshop
8. Bradley AR, Manna Z, Sipma HB (2005) What's Decidable About Arrays? In VMCAI. Springer
9. Brillout A, Kroening D, Rümmer P, Wahl T (2011) Beyond Quantifier-Free Interpolation in Extensions of Presburger Arithmetic. In VMCAI. Springer
10. Cadar C, Sen K (2013) Symbolic execution for software testing: three decades later. Communicat ACM 56(2):82–90
11. Cha SK, Avgerinos T, Rebert A, Brumley D (2012) Unleashing mayhem on binary Code. In S &P 2012
12. Chakraborty S, Meel K, Mistry R, Vardi M (2016) approximate probabilistic inference via Word-Level counting. AAAI 30(1)
13. Cimatti A, Clarke EM, Giunchiglia F, Roveri M (1999) NuSMV: a new Symbolic Model Verifier. In CAV'99. Springer
14. Clarke E, Kroening D, Lerda F (2004) A Tool for Checking ANSI-C Programs. In TACAS. Springer
15. Clarke EM, Emerson EA (1982) Design and synthesis of synchronization skeletons using branching time temporal logic. In Logics of Programs. Springer
16. Clarkson MR, Finkbeiner B, Koleini M, Micinski KK, Rabe MN, Sánchez C (2014) Temporal logics for hyperproperties. In principles of security and trust. Springer
17. Clarkson MR, Schneider FB (2010) Hyperproperties. J Comput Secur 18(6):1157–1210
18. Cook B, Podelski A, Rybalchenko A (2006) Terminator: Beyond Safety. In: CAV. Springer
19. Cowan C, Pu C, Maier D, Walpole J, Bakke P, Beattie S, Grier A, Wagle P, Zhang Q, Hinton H (1998) StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In USENIX Security
20. Daniel LA, Bardin S, Rezk T (2020) Binsec/Rel: Efficient relational symbolic execution for constant-time at binary-level. In S &P 2020. IEEE
21. David R, Bardin S, Ta TD, Mounier L, Feist J, Potet ML, Marion JY (2016) BINSEC/SE: A dynamic symbolic Execution toolkit for binary-level analysis. In SANER 2016. IEEE
22. David R, Bardin S, Feist J, Mounier L, Potet ML, Ta TD, Marion JY (2016) Specification of concretization and symbolization policies in symbolic execution. In ISSTA 2016. ACM
23. de Moura L, Bjørner N (2008) Z3: An Efficient SMT Solver. In TACAS. Springer
24. Djoudi A, Bardin S (2015) BINSEC: Binary code analysis with low-level regions. In: TACAS. Springer
25. Farinier B (2020) Decision Procedures for Vulnerability Analysis. Ph.D. thesis, Université Grenoble-Alpes

26. Farinier B, Bardin S, Bonichon R, Potet ML (2018) Model generation for quantified formulas: a taint-based approach. In CAV. Springer
27. Farinier B, David R, Bardin S, Lemerre M (2018) Arrays made simpler: an efficient, scalable and thorough Preprocessing. In LPAR-22
28. Ge Y, de Moura L (2009) Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In CAV. Springer
29. Girol G, Farinier B, Bardin, S (2021) Not all bugs are created equal, but robust reachability can tell the difference. In CAV '21. Springer
30. Godefroid P (2011) Higher-order test generation. In PLDI '11. ACM
31. Godefroid P, Klarlund N, Sen K (2005) DART: directed automated random testing. In PLDI 2005. ACM
32. Godefroid P, Levin MY, Molnar D (2012) SAGE: Whitebox fuzzing for security testing: SAGE has had a remarkable impact at Microsoft. Queue 10(1):20–27
33. Goguen JA, Meseguer J (1982) Security Policies and Security Models. In S &P 1982. IEEE
34. Gomes CP, Sabharwal A, Selman B (2008) Model Counting. In Handbook of Satisfiability. Ios press edn
35. Hansen T, Schachte P, Søndergaard H (2009) State Joining and Splitting for the Symbolic Execution of Binaries. In Runtime Verification. Springer
36. Hansson H, Jonsson B (1994) A logic for reasoning about time and reliability. Formal Aspects o Comput 6(5):512–535
37. Hart S, Sharir M, Pnueli A (1983) Termination of Probabilistic Concurrent Program. ACM Transact Program Lang Syst 5(3):356–380
38. Heelan S (2009) Automatic generation of control flow hijacking exploits for software vulnerabilities. Master's thesis, University of Oxford
39. Heusser J, Malacaria P (2010) Quantifying information leaks in software. In ACSAC '10. ACM Press
40. Holler C, Herzig K, Zeller A (2012) Fuzzing with code fragments. In 21st USENIX security symposium. USENIX Association
41. Kim S, McCamant S (2018) Bit-vector Model counting using statistical estimation. In TACAS. Springer
42. Kurpiewski D, Knapik M, Jamroga W (2019) On Domination and Control in Strategic Ability. AAMAS .9
43. Laroussinie F, Meyer A, Petonnet E (2010) Counting CTL. In: Foundations of Software Science and Computational Structures. pp. 206–220. Lecture Notes in Computer Science
44. Livshits B, Sridharan M, Smaragdakis Y, Lhoták O, Amaral JN, Chang BYE, Guyer SZ, Khedker UP, Møller A, Vardoulakis D (2015) In defense of soundiness: A manifesto. Communicat ACM 58(2):44–46
45. de Moura L, Bjørner N (2007) Efficient E-Matching for SMT Solvers. In Automated Deduction - CADE-21. Springer
46. Niemetz A, Preiner M, Biere A (2015) Boolector 2.0: System description. J Satisf Boolean Modeling Comput 9(1)
47. O'Hearn PW (2020) Incorrectness logic. POPL.4:1–32
48. Recoules F, Bardin S, Bonichon R, Mounier L, Potet ML (2019) Get Rid of Inline assembly through verification-oriented lifting. In ASE 2019. IEEE
49. Reynolds A, Tinelli C, Goel A, Krstić S (2013) Finite model finding in SMT. In CAV. Springer
50. Sen K, Marinov D, Agha G (2005) CUTE: a concolic unit testing engine for C. In ESEC/FSE-13. ACM
51. Shoshitaishvili Y, Wang R, Salls C, Stephens N, Polino M, Dutcher A, Grosen J, Feng S, Hauser C, Kruegel C, Vigna G (2016) SOK: (State of) The art of war: Offensive techniques in binary analysis. In: SP 2016
52. Song F, Touili T (2014) Efficient CTL model-checking for pushdown systems. Theor Comput Sci 549:127–145
53. Urban C, Miné A (2017) Inference of ranking functions for proving temporal properties by abstract interpretation. Comput Lang Syst Struct 47:77–103
54. Urban C, Ueltschi S, Müller P (2018) Abstract interpretation of CTL properties. In SAS 2018. Springer
55. Williams N, Marre B, Mouy P, Roger M (2005) Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In EDCC-05. Springer