Inference of Robust Reachability Constraints

Yanis Sellami, Guillaume Girol, Frédéric Recoules, Damien Couroussé, Sébastien Bardin

Inference of Robust Reachability Constraints

YANIS SELLAMI, Univ. Grenoble Alpes - CEA - List, France
GUILLAUME GIROL, Université Paris-Saclay - CEA - List, France
FRÉDÉRIC RECOULES, Université Paris-Saclay - CEA - List, France
DAMIEN COUROUSSÉ, Univ. Grenoble Alpes - CEA - List, France
SÉBASTIEN BARDIN, Université Paris-Saclay - CEA - List, France

Characterization of bugs and attack vectors is in many practical scenarios as important as their finding. Recently, Girol *et al.* have introduced the concept of *robust reachability*, which ensures a perfect *reproducibility* of the reported violations by distinguishing inputs that are under the control of the attacker *(controlled inputs)* from those that are not *(uncontrolled inputs)*, and proposed first automated analysis for it. While it is a step toward distinguishing severe bugs from benign ones, it fails for example to describe violations that are *mostly* reproducible, i.e., when triggering conditions are likely to happen, meaning that they happen for all uncontrolled inputs but a few corner cases. To address this issue, we propose to leverage theory-agnostic abduction techniques to generate *constraints on the uncontrolled program inputs that ensure that a target property is robustly satisfied*. Our proposal comes with an extension of robust reachability that is generic on the type of trace property and on the technology used to verify the properties. We show that our approach is complete *w.r.t.* its *inference language*, and we additionally discuss strategies for the efficient exploration of the inference space. We demonstrate the feasibility of the method and its practical ability to refine the notion of robust reachability with an implementation that uses robust reachability oracles to generate constraints on standard benchmarks from software verification and security analysis. We illustrate the use of our implementation to a vulnerability characterization problem in the context of fault injection attacks. Our method overcomes a major limitation of the initial proposal of robust reachability, without complicating its definition. From a practical view, this is a step toward new verification tools that are able to characterize program violations through high-level feedback.

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

Additional Key Words and Phrases: program analysis, abduction, precondition inference, symbolic execution

ACM Reference Format:
Yanis Sellami, Guillaume Girol, Frédéric Recoules, Damien Couroussé, and Sébastien Bardin. 2024. Inference of Robust Reachability Constraints. *Proc. ACM Program. Lang.* 8, POPL, Article 91 (January 2024), 30 pages. https://doi.org/10.1145/3632933

1 INTRODUCTION

Formal verification techniques, such as symbolic execution [Cadar and Sen 2013] or bounded model checking [Clarke et al. 2004], are popular approaches to detect bugs and vulnerabilities in programs.

Authors' addresses: Yanis Sellami, Univ. Grenoble Alpes - CEA - List, F-38000 Grenoble, France, yanis.sellami@cea.fr; Guillaume Girol, Université Paris-Saclay - CEA - List, F-91120 Palaiseau, France, guillaume.girol.2014@polytechnique.org; Frédéric Recoules, Université Paris-Saclay - CEA - List, F-91120 Palaiseau, France, frederic.recoules@cea.fr; Damien Couroussé, Univ. Grenoble Alpes - CEA - List, F-38000 Grenoble, France, damien.cousousse@cea.fr; Sébastien Bardin, Université Paris-Saclay - CEA - List, F-91120 Palaiseau, France, sebastien.bardin@cea.fr.

ACM 2475-1421/2024/1-ART91
https://doi.org/10.1145/3632933

Yet, characterization of bugs and attack vectors is in many practical scenarios as important as their finding, as in many situation an expert has to examine the reported violation for further actions (*e.g.*, deciding whether it must be fixed or not). Unfortunately, the aforementioned techniques fail to provide a clear characterization of the detected violations—for example they cannot characterize how easily an attacker could trigger a reported vulnerability. While it is possible in principle to apply counting algorithms [Aziz et al. 2015; Darwiche 2000; Fremont et al. 2017; Gomes et al. 2008; Kim and McCamant 2018; Lagniez and Marquis 2019] to improve the understanding of a vulnerability, such techniques remain costly and their results are difficult to exploit, typically for vulnerability comparisons.

Problem. Recently, Girol *et al.* have introduced the concept of *robust reachability* [Girol et al. 2021, 2022] which ensures that some user-*controlled* input permits a perfect *reproducibility* of the reported violations, independently of the values of the other *(uncontrolled)* inputs, and proposed first automated analysis for it. While it is a step toward distinguishing severe bugs from benign ones, it fails to describe violations that are *mostly* reproducible. For instance, a violation that can be triggered for all but a few uncontrolled values or a violation requiring a specific but probable relation on the uncontrolled input, are considered non robustly reachable, even though an attacker could still fairly easily exploit them.

Goal. Our general goal is to build on the initial proposal of robust reachability by Girol *et al.* to make it more flexible and thus more practical, by capturing more realistic scenarios. One natural direction is to define a quantitative version of robust reachability, yet it requires advanced (and expensive) model counters [Bardin and Girol 2022], and the feedback reported to the user is minimalist (a mere value or ratio).

We follow here another direction. Robust reachability can actually accommodate assumptions on the uncontrolled inputs, allowing to express for example that a given controlled input a_0 will indeed succeed as long as uncontrolled input x_0 is even. Such assumptions can be seen as a logical description of constraints on uncontrolled inputs that suffice to trigger a target vulnerability. This description is easier to understand for the security export and can be forwarded to other tools that handle logic constraints to automatically obtain qualitative comparison information such as inclusion or equivalence. If the assumption is simple enough, it is also possible to perform an easy approximate counting of the number of encompassed program inputs. Yet, it is currently up to the user to provide such assumptions. We want to design an automatic method able to infer such robust reachability constraints. By doing so, we hope to make robust reachability more practical, while retaining its qualitative flavor. Ideally, this should also permit to obtain a method that is independent of the type of properties considered, and of the practical verification technique used.

Proposal. We propose to leverage generic formula abduction techniques [Josephson and Josephson 1994] to generate conditions on the program input that ensure that a given trace property for a program is *robustly satisfied*. This is done alongside a straightforward extension of robust reachability to more general program trace properties.

Abduction, in the general case, is the generation of a missing hypothesis that, when added to the set of axioms, permits to deduce a previously unprovable goal. Abduction techniques targeting the generation of program contracts are usually based on concrete rule definitions to compute such hypotheses [Albarghouthi et al. 2016; Calcagno et al. 2009]. This restriction means that such techniques are dedicated to a given type of program definition, a given type of property and a given type of hypotheses. These restrictions imply that any such system needs to be adapted to handle any variation of definitions it initially targeted.

For first-order logic formulas, contradiction-based abductive reasoning [Echenim et al. 2018; Reynolds et al. 2020] use an input inference language from which they build candidate solutions that are verified against an abstract oracle. This setup makes it possible to remain independent from the theory in which the formulas are expressed, as well as from the technology used to verify that such formulas are correct solutions to the abduction problem (SMT-Solver, Resolution prover, *etc.*), and hence to benefit directly from advances to these technologies. We adapt such generic formula-level contradiction-based abduction techniques to program trace satisfiability in order to generate *sufficient* preconditions.

Contributions. We claim the following contributions:

- We propose the first program-level abduction algorithm (Algorithm 1, Section 4.3) tailored to robust trace property satisfaction, an extension of robust reachability. Given an inference language, a program, a trace property, and assuming the existence of oracles for verifying robust trace properties under assumption, we build a sufficient conditions on a program uncontrolled inputs we can complement with a controlled input that ensures the unconditional trigger of the target trace property. The technique is agnostic *w.r.t.* to these oracles, and we clarify its properties (Theorem 4.9);
- We design dedicated pruning techniques to ensure that the computation time remains affordable (Section 5), and we prove that our optimized algorithm is correct and complete *w.r.t.* the inference language (Theorem 4.9, Theorem 5.2) as long as oracles are;
- We implement a specialization of our algorithm to robust reachability of program locations (Section 6.1, Section 6.2) and evaluate it on standard benchmarks from software verification [Beyer 2012] and security analysis [Dureuil et al. 2016], demonstrating the feasibility of the method and its practical ability to refine the notion of robust reachability (Section 6.4);
- We propose an application to a realistic security evaluation scenario where we study the characterization of vulnerabilities introduced by fault injection attacks on typical implementations of a security primitive (Section 6.5). We show that our method permits a relevant security evaluation that outperforms previous methods, both in the quantity of characterized vulnerabilities and in the quality of the characterizations.

Interestingly, from a theoretical view, our approach overcomes a major limitation of robust reachability *without complicating its definition*, without introducing any expensive quantitative reasoning. This is achieved by a new theory-agnostic abduction procedure that differs from previous work in that it exploits necessary constraints and can solve problems with (a subset of) universally quantified variables. Moreover, from a practical view, this paves the way to new verification tools able to better characterize program violations by providing high-level feedback.

2 MOTIVATING EXAMPLE

Let us consider the synthetic code sample in Figure 1. Here, the function `kmemcpy` tries to take advantage of memory alignment to improve the copy efficiency—e.g., by using vectorized instruction set such as SSE for `x86` or `NEON` for `ARM`. Thus, when both the source and destination pointers, `src` and `dst`, and the buffer size `n` are multiples of 32, the function copies memory bytes by chunks of the same amount.

Buffer overflow. However, this so-called *fast path* suffers from an implementation error: the loop does an additional round since '<=' is used instead of '<' at line 7. This way, 32 input bytes may be unexpectedly copied beyond the size of the destination buffer. Such buffer overflow can be especially harmful when the destination area is allocated on the stack—an attacker may overwrite the return address of the function to which the buffer belongs and change the execution paths.

```
1  typedef struct { unsigned char bytes[32]; } uint256_t;
2
3  void kmemcpy (void *dst, const void *src, size_t n)
4  {
5    if ((((intptr_t)dst | (intptr_t)src | n) & 0b11111)) // slow path
6      for (size_t i = 0; i < n; i += 1) *((uint8_t*)dst + i) = *((uint8_t*)src + i);
7    else // fast path
8      for (size_t i = 0; i <= (n >> 5); i += 1) *((uint256_t*)dst + i) = *((uint256_t*)src + i);
9  }
```

Fig. 1. Example of a faulty optimized memcpy function, vulnerable to non-deterministic buffer overflow

```
1  void f(const char *input)
2  {
3    char buf[64] = { 0 };
4    kmemcpy(buf, input, 64); buf[63] = 0;
5    puts(buf);
6    return;
7  }
8
9  void g() { puts("hijacked"); exit(-1); }
```

```
1  int main (int argc, char *argv[])
2  {
3    char src[96] = "doing things";
4    for (int i = 0; i < 32; i += sizeof(void (*)()))
5      *(void (**)())(src + 64 + i) = &g;
6    f(src);
7    return 0;
8  }
```

Fig. 2. Example of a non-deterministic buffer overflow exploitation over the kmemcpy function of 1

An example of such an attack is given in Figure 2. Here, by carefully crafting the content of the buffer `input` given to the function `f`, the attacker can jump at a chosen address (the address of the hijacking function g) without any direct call to it. More precisely, the content of the *controlled* buffer `src` is crafted to overwrite the return address of `f` by the address g; the loop in the `main` at line 16 initializes 32 extra bytes with the value &g. These bytes can be copied by the function kmemcpy beyond the buffer `buf` where the call instruction is pushing the return address on the stack. However, the memory alignment of buffers is not *controlled* by the attacker and varies in each runs. Thus, the buffer overflow does not occur on each execution and the program can return both `0` (normal execution) or `-1` (reaching g).

Reachability. We can express the vulnerability to control flow hijacking as a reachability property: when the **`return`** statement is reached from an address that is not the caller's address. Symbolic execution tools [Cadar and Sen 2013] automate the query resolution. In particular, they provide a concrete assignment of the input memory state for which the program reaches the requested target, actually proving its feasibility. For instance, symbolic execution may return that a vulnerability is reachable with the following assignment: `&buf == 0xffffcc80`, `&input == 0xffffcd40` and `input[80..83] == {0xde, 0xad, 0xbe, 0xef}`.

However, symbolic execution does not give insight on whether the target vulnerability is easy to reach or not. Indeed, having a concrete example is not enough to characterize the dangerous assignments because we do not know if the values the analysis returned are mandatory nor if there exist relations between them.

Robust reachability. To address this issue, Girol *et al.* have introduced the concept of *robust reachability* [Girol et al. 2021] (recalled here in Section 3.3). Their framework allows to separate the addresses of the input memory state between the ones an attacker can craft (here: the content of `input`), called *controlled variables*, and the ones it cannot (here: the buffer addresses `&input` and `&buf`), called *uncontrolled variables*. The framework looks for a concrete assignment on controlled variables that reaches a given program state whatever the values of the uncontrolled ones. This permits to distinguish vulnerabilities that are replicable from those that are not. We believe that a replicable attack is more dangerous than an attack that is not and the framework of robust reachability is able to make the distinction without the use of counting procedures.

Here, the buffer overflow is never always reachable for a given content of `input`. For instance, it will not happen if the buffer address `&buf` is `0xffffcc88`. The framework then concludes that it is not robustly reachable, leaving the above question open.

Robust reachability under assumption. For now, we know that there exists *at least* an assignment leading to the buffer overflow, but also *at least* one that will not. Yet, we are actually looking for a way to characterize how dangerous assignments are, without enumerating them.

This is where abduction comes into play. We try to generate a necessary and sufficient precondition over the state of the uncontrolled variables that makes the program state under study robustly reachable. In short, our approach iteratively builds and tests several precondition candidates in order to find an assumption such that all the compliant assignments lead to the buffer overflow. It turns out that taking the *fast path* is both necessary and sufficient to override the return address of function `f`. Our approach generates the constraint (`&buf % 32 == 0` ∧ `&input % 32 == 0`), under which the buffer overflow is robustly reachable with a model that sets `input[80..83]` to the address of the hijacking function, such as *e.g.*, `input[80..83] == {0xde, 0xad, 0xbe, 0xef}`.

This helps to understand what is the source of the nondeterminism of the attack and gives some hints on the likeliness of its success. We then conclude that, although not robust, the vulnerability is still serious and likely to be exploitable.

Targeted properties. We are primarily interested in properties relevant for software-level security. The inference framework we propose considers a large class of reachability properties, as it builds upon *oracles* for the targeted properties—and we will discuss the quality of the framework depending on the quality of the oracles. Yet, our current implementation itself is restricted to *location* reachability, that we see as a good trade-off between implementation simplicity and practical expressiveness, as *local assertion* reachability (*e.g.*, buffer overflows), *finite trace* reachability (e.g., use-after-free bugs) or even *k-hyper-reachability* (*e.g.*, information leakage) can be reduced to it through proper program instrumentation—respectively with extra `assert`, monitors or self-composition.

Regarding controlled variables, note also that, as long as they cover an under-approximation of what is controllable in reality, a robust symbolic execution (or any other under-approximation of robust reachability) will return correct witnesses.

3 BACKGROUND AND NOTATIONS

We present a number of proof techniques for verifying trace properties, robust trace properties, and the related definitions.

3.1 Notations

Let us consider a set of variables $\mathcal{A}$ and a set of values $\mathcal{V}$. We call memory state s a total mapping from variables to values, and denote by $\mathcal{S}_{\mathcal{A}}$ the set of all possible memory states on $\mathcal{A}$. Variables are understood in a general, abstract way: they encompass all the possible inputs, including memory addresses, registers, etc. We call program a deterministic transition function $\rightarrow_P \in \mathcal{S}_{\mathcal{A}} \times \mathcal{S}_{\mathcal{A}}$ and denote by $\rightarrow_P^*$ its transitive reflexive closure. We assume that we can deterministically obtain a program location from the memory state, typically by extracting the program counter. We call trace a sequence t of states of $\mathcal{S}_{\mathcal{A}}$ and path a sequence π of program locations. We say that a trace t follows the path π if both sequences have the same length and if for any sequence index i, the program location of $t[i]$ is equal to $\pi[i]$. Given a program $\rightarrow_P$, we denote by P^* the set of all program traces. We assume traces to be deterministic, that is, from an initial memory state s, there is one and only one trace $t \in P^*$ such that $t[0] = s$, and denote this trace $\rightarrow_P^*(s)$. Given a bound k, we additionally denote by P^k the set of program traces of maximal length k. For simplicity, we identify the formal inputs to the program as a variable y on the memory states of $\mathcal{S}_{\mathcal{A}}$. Given a

program $\rightarrow_P$, we consider first-order predicates [Caferra 2013] on traces and on memory states. We denote by $\phi[s]$ the evaluation of a formula ϕ for a memory state s. We consider the standard notion of satisfiability and we denote logical entailments by $\models$. For simplicity, we assume that we only consider predicates for which entailment is decidable. We will discuss what happens if this is not the case in Section 4.4. In all the following, we consider oracles for trace property satisfaction and robust trace property satisfaction, denoted by $O^{\exists\exists}$ and $O^{\exists\forall}$ respectively, the definition of which is detailed in Section 3.4. We denote by $\mathcal{A}_C$ the set of controlled variables (Section 3.3).

3.2 Trace Properties

With these notations, we can define properties on program traces.

Definition 3.1 (trace property satisfaction). Given a program $\rightarrow_P$ and a predicate on traces ψ, we say that $\rightarrow_P$ *satisfies* ψ if $\exists t \in P^*, \psi[t]$.

Note that it is often useful to restrict the length of program traces up to a bound k. In such a case, we can say that $\rightarrow_P$ k-satisfies ψ if $\exists t \in P^k, \psi[t]$. In all the following, we work without such a bound but note that the properties can be restricted for traces of at most k memory states. This can typically be used for reducing computation time.

As we aim to constrain program executions, it is useful to consider the satisfaction of trace properties under an additional condition on program inputs.

Definition 3.2 (trace property satisfaction under assumption). Given a program $\rightarrow_P$, a predicate on traces ψ and a predicate ϕ on memory states. We say that $\rightarrow_P$ *satisfies* ψ *under the assumption* ϕ if $\exists t \in P^*, \psi[t] \wedge \phi[t[0]]$. Similarly, we say that $\rightarrow_P$ *k-satisfies* ψ *under the assumption* ϕ if $\exists t \in P^k, \psi[t] \wedge \phi[t[0]]$.

3.3 Robust Trace Properties

In the context of security however, the standard concept of trace property satisfaction may not be enough to understand the severity of the vulnerability that has been found. In order to differentiate between vulnerabilities that an attacker can trigger reliably from those he cannot, Girol *et al.* have introduced the notion of robustness [Girol et al. 2021]. The core idea is to separate the program inputs between the ones an attacker can choose, called controlled inputs, and the others, called uncontrolled inputs. One then looks for traces satisfying the property for at least one value of the controlled inputs but independently of the value of the uncontrolled inputs. Their approach encompasses an important aspect of exploitability: replicability of the found bugs and their exploitation in attacks. We follow the hypothesis that an attack that is replicable is far more dangerous than an attack that is not. Robustness offers a way to perform such reasoning without resorting to expensive counting problems.

Formally, we consider the memory states $\mathcal{S}_{\mathcal{A}_C}$ of a subset of variables $\mathcal{A}_C \subset \mathcal{A}$ where $\mathcal{A}_C$ represents the variables of the initial state that the attacker can control.

Definition 3.3 (robust trace property satisfaction). Given a program $\rightarrow_P$, a predicate on traces ψ and a set of controlled variables $\mathcal{A}_C$, we say that $\rightarrow_P$ *robustly satisfies* ψ if we can find a value of the controlled variables for which the predicate ψ is satisfied independently of the value of the other variables, that is, if we have $\exists a \in \mathcal{S}_{\mathcal{A}_C}, \forall t \in P^*, t[0]|_{\mathcal{A}_C} = a \Rightarrow \psi[t]$, where $t[0]|_{\mathcal{A}_C}$ denotes the restriction of the memory state $t[0]$ to the variables of $\mathcal{A}_C$. We call this a a *witness*.

Similarly to what we introduced for trace properties, we can look for trace properties that are robust under a restriction on the program inputs.

Definition 3.4 (robust trace property satisfaction under assumption). Given a program $\rightarrow_P$, a predicate on traces ψ and a set of controlled variables $\mathcal{A}_C$, consider a predicate on memory states ϕ. We say that $\rightarrow_P$ *robustly satisfies* ψ *under* ϕ if $\exists a \in \mathcal{S}_{\mathcal{A}_C}, (\exists t \in P^*, t[0]|_{\mathcal{A}_C} = a \wedge \phi[t[0]]) \wedge \forall t \in P^*, (t[0]|_{\mathcal{A}_C} = a \wedge \phi[t[0]]) \Rightarrow \psi[t]$

Proposition 3.5. (1) *Robust trace property satisfaction (resp. under assumption) implies property satisfaction (resp. under assumption). The converse does not hold.*
(2) *When $\mathcal{A}_C = \mathcal{A}$, robust trace property satisfaction (resp. under assumption) is equivalent to property satisfaction (resp. under assumption).*

Another concept that is usually defined for program verification analysis techniques is program assertions. While assumptions and assertions are equivalent in the non-robust case, the presence of a universally quantified controlled memory introduces a disparity between the two notions. In this framework, assertions can be produced in two ways: either by instrumenting the transition function $\rightarrow_P$ with an additional test representing the assertion, or by verifying a trace property constructed as the conjunction of the original property with the assertion.

3.4 Trace Properties Oracle

We assume the existence of oracles for verifying trace properties satisfaction under assumption and robust trace properties satisfaction under assumption. Oracles are useful abstractions for the design of an algorithmic framework that is agnostic both on the type of the programs we consider and on the type of trace-properties. Note that these oracles can be restricted to a subclass of trace properties. We denote the oracle for trace property satisfaction under assumption (Definition 3.2) by $O^{\exists\exists}(\rightarrow_P, \psi, \phi)$ and for robust trace property satisfaction under assumption (Definition 3.4) by $O^{\exists\forall}(\rightarrow_P, \mathcal{A}_C, \psi, \phi)$. We assume that, when $O^{\exists\exists}$ terminates, it returns a pair $b, s \in \mathbb{B} \times \mathcal{S}_{\mathcal{A}}$ where b is the decision of the oracle and s is a witness of the decision used when $b = \top$. Similarly, we assume that, when $O^{\exists\forall}$ terminates, it returns a pair $b, a \in \mathbb{B} \times \mathcal{S}_{\mathcal{A}_C}$ where b is the decision of the oracle and a is a witness of the decision used when $b = \top$. Note that for both oracle types, when $b = \bot$, the witness has no relevance and will not be used. In order to simplify the notations, we will abusively consider that these oracles simply return the Boolean when we do not wish to use the witness.

Definition 3.6 (oracle correctness). An oracle for satisfaction $O^{\exists\exists}$ is *correct* when for any program $\rightarrow_P$, trace property ψ, and assumption ϕ, $O^{\exists\exists}$ terminates and $O^{\exists\exists}(\rightarrow_P, \psi, \phi) = \top, s$ implies that $\rightarrow_P$ satisfy ψ under ϕ and $\psi[\rightarrow_P^* (s)]$. Additionally, $O^{\exists\exists}$ is correct for a given subset of programs, trace properties and assumptions if this proposition is satisfied for any triplet in this subset.

An oracle for robust satisfaction $O^{\exists\forall}$ is *correct* when for any program $\rightarrow_P$, set of controlled variables $\mathcal{A}_C$, trace property ψ and assumption ϕ, $O^{\exists\forall}$ terminates and $O^{\exists\forall}(\rightarrow_P, \mathcal{A}_C, \psi, \phi) = \top, a$ implies that $\rightarrow_P$ robustly satisfies ψ under ϕ with witness a. Additionally, $O^{\exists\forall}$ is correct for a given subset of programs, sets of controlled variables, trace properties and assumptions if this proposition is satisfied for any quadruplet in this subset.

Definition 3.7 (oracle completeness). An oracle for satisfaction $O^{\exists\exists}$ is *complete* when for any program $\rightarrow_P$, trace property ψ and assumption ϕ, $O^{\exists\exists}$ terminates and $O^{\exists\exists}(\rightarrow_P, \psi, \phi) = \bot, s$ implies that $\rightarrow_P$ does not satisfy ψ under ϕ. Additionally, $O^{\exists\exists}$ is complete for a given subset of programs, trace properties and assumptions if this proposition is satisfied for any triplet in this subset.

An oracle for robust satisfaction $O^{\exists\forall}$ is *complete* when for any program $\rightarrow_P$, set of controlled variables $\mathcal{A}_C$, trace property ψ and assumption ϕ, $O^{\exists\forall}$ terminates and $O^{\exists\forall}(\rightarrow_P, \mathcal{A}_C, \psi, \phi) = \bot, t$ implies that $\rightarrow_P$ does not robustly satisfy ψ under ϕ. Additionally, $O^{\exists\forall}$ is complete for a given

subset of programs, sets of controlled variables, trace properties and assumptions if this proposition is satisfied for any quadruplet in this subset.

An oracle is an under-approximation of the property satisfaction if it is correct but incomplete. It is an over-approximation if it is complete but incorrect. Note also that a non-terminating correct oracle can be straightforwardly converted into a terminating correct oracle using a timeout and by returning $\bot$ in case the timeout was triggered. The same can be done for a non-terminating complete oracle by returning $\top$ instead.

Example. An example of usual trace property with the corresponding oracles is program location k-reachability. Symbolic execution [Cadar and Sen 2013] is an algorithmic proof method for reachability problems. It broadly works by expressing k-reachability properties as *quantifier-free* SMT constraints, the solving of which is delegated to SMT solvers [Barrett et al. 2009] such as Z3 [de Moura and Bjørner 2008] or cvc5 [Barbosa et al. 2022]. Symbolic execution is both correct and complete for k-reachability as long as the theory used to express reachability constraints remains decidable—which is usually the case.

Symbolic execution has been extended to robust symbolic execution [Girol et al. 2021, 2022] for handling robust k-reachability properties. This algorithm relies on satisfiability requests for *universally quantified* formulas, which is more expensive. Robust symbolic execution is similarly correct and complete for robust k-reachability as long as the universally quantified theory in which the reachability constraints are expressed remains decidable.

For both symbolic execution and robust symbolic execution, assumptions and assertions can be introduced in the generated path predicates. We also point out that both standard and robust symbolic executions can be extended to target other types of properties *s.a.* safety [Girol et al. 2022] or hyper safety [Daniel et al. 2020].

3.5 Abduction

Abduction [Josephson and Josephson 1994] is usually defined as the search for missing preconditions which entail an unexplained goal. More precisely, given a hypothesis ϕ_H and a goal ϕ_G that is not a logical consequence of ϕ_H, abduction is the process of computing an additional hypothesis ϕ_M that is consistent with ϕ_H and such that ϕ_G is a logical consequence of $\phi_H \wedge \phi_M$. Additionally, the missing hypothesis ϕ_M should be as general as possible. The search for ϕ_M can be further restricted to expressions of a given inference language.

- Abductive inference has been successfully used in a large number of software-based applications such as compositional program verification [Calcagno et al. 2009], specification synthesis [Albarghouthi et al. 2016], knowledge base extensions [Koopmann et al. 2020] and loop invariants generation [Dillig et al. 2013; Echenim et al. 2019]. For program verification, abduction techniques [Albarghouthi et al. 2016; Calcagno et al. 2009] are defined as *white-box computation frameworks*, which means that one needs to adapt such methods for any distinct type of program and trace property they want to verify;
- Recent research on *formulas abduction* has focused on the design of *theory-agnostic algorithms* [Echenim et al. 2018; Reynolds et al. 2020] that can be used as is when the problem at hand can be expressed in first-order logic with no additional restriction. Such methods can be applied independently of the theory used and of the type of solver employed to decide such properties, but they cannot be applied straightforwardly for the abduction of program properties.

In this work, we propose to adapt theory-agnostic formulas abduction techniques to handle the abduction of program constraint with a similar level of genericity, that is, to obtain an algorithmic framework

that is generic with respect to the type of program, the type of program properties and the technology of the property verifier.

4 ABDUCTIVE INFERENCE OF ROBUST TRACE PROPERTY CONSTRAINTS

We present in this section an algorithmic framework for the abductive inference of robust trace property constraints. We adapt theory-agnostic abduction techniques for handling varying verification conditions through oracle calls. The core idea is to explore a language of candidate solutions and to check these candidates with robust oracle calls. Given an inference language $\mathcal{G}$, a program $\rightarrow_P$, a trace property ψ and a set of controlled variables $\mathcal{A}_C$, the baseline algorithm (Algorithm 1) iterates over the candidates of $\mathcal{G}$ and uses a robust oracle call to check if a given candidate is a solution. The method maintains a set of solutions and iteratively simplifies it each time a new solution is found to keep a minimal set of solutions (see Section 4.2). If the set of generated solutions is found to be necessary for the satisfaction of ψ, the algorithm can stop the exploration of $\mathcal{G}$ prematurely. We prove that, when the robust oracle has the expected correctness and completeness property, our algorithm is correct and complete *w.r.t.* its inference language $\mathcal{G}$, and returns the minimal set of solutions in $\mathcal{G}$. This baseline algorithm is simplistic and inefficient, but permits to define the properties of the framework (Theorem 4.9) from which we can derive the results on the final algorithm in Section 5 (Theorem 5.2). As this baseline solution remains inefficient, we describe in Section 5 a collection of strategies to make the technique viable in practice.

Inference Language. As previously mentioned, our algorithmic framework for the generation of robust trace property constraints is based on the exploration of a constraints inference language provided as an input to the algorithm.

Definition 4.1 (inference language). We call *inference language* a finite set $\mathcal{G}$ of first-order predicates on memory states. In all the following, we call *candidate* any predicate on memory states $\phi \in \mathcal{G}$. Moreover, if ϕ is a sufficient robust trace property constraint for the target problem, we will say that ϕ is a *solution*.

In practice, inference languages are usually built from applying Boolean connectors to a finite set of literals built from program inputs (*e.g.*, variables or memory accesses) and adequate operators (*e.g.*, linear arithmetic). Note that the following algorithmic construction does not require the inference language to be closed by conjunction, although some of the optimization strategies are easier to apply when this is the case. Note also that the inference language is thought to be program-dependent, as, *e.g.*, addresses corresponding to program variables and literal values from which we can obtain meaningful trace property constraints will differ from program to program. See, *e.g.*, Section 6.3, for an example of concrete inference language definition.

Goal. Our goal is, given a program $\rightarrow_P$, an inference language $\mathcal{G}$, a trace property ψ and a set of controlled variables $\mathcal{A}_C$, to automatically generate sufficient robust trace property constraints (Definition 4.2). This is a problem of abduction on trace property constraints. Our work aims to use practical refinements that can improve verification time for distinct program explorations, which usually leads to variating verification schemes for each tested candidate. This variation of the verification condition is out of the scope of the aforementioned abduction techniques. Strategies for the efficient implementation of this algorithmic framework will be presented in Section 5.

We propose an algorithm, named ARCInfer, presented as Algorithm 2 in Section 5.4, that builds on the baseline algorithmic framework presented as Algorithm 1 in Section 4.3. Assuming correct and complete oracles, our algorithm is correct, complete, and minimal *w.r.t.* to an input inference language. Our algorithm additionally generates a weakest robust trace-property constraint (Definition 4.6) if it can be built from the inference language.

4.1 Preliminary Definitions

We start by providing a definition for *sufficient* trace property predicates over input memory states, which describes constraints on inputs under which a trace property is guaranteed to be satisfied.

Definition 4.2 (sufficient robust trace property constraint). Given a program $\rightarrow_P$, a predicate on traces ψ, a set of controlled variables $\mathcal{A}_C$, *a predicate on memory states ϕ is a sufficient robust constraint (with a witness a)* if $\rightarrow_P$ robustly satisfies ψ under ϕ (with the witness a).

Definition 4.3 (necessary trace property constraint). Given a program $\rightarrow_P$, a predicate on traces ψ, a set of controlled variables $\mathcal{A}_C$, *a predicate on memory states ϕ is a necessary trace property constraint for ψ* if $\forall t \in P^*, \phi[t[0]] \vee \neg\psi[t]$

Proposition 4.4. *Given a program $\rightarrow_P$, a predicate on traces ψ, a set of controlled variables $\mathcal{A}_C$. If ϕ_s is a sufficient robust constraint for ψ, with a witness a and ϕ_n is a necessary constraint for ψ, then $\forall s \in \mathcal{S}_{\mathcal{A}}, s|_{\mathcal{A}_C} \neq a \vee \neg\phi_s[s] \vee \phi_n[s]$.*

Proof. If this is not the case, the we have a memory state $s \in \mathcal{S}_{\mathcal{A}}$ such that $s|_{\mathcal{A}_C} = a \wedge \phi_s[s] \wedge \neg\phi_n[s]$. As ϕ_s is a sufficient robust constraint for ψ with the witness a, we have from Definition 3.4 that $\psi[\rightarrow_P^*(s)]$. Moreover, as ϕ_n is a necessary constraint for ψ, we have from Definition 4.3 that $\neg\psi[\rightarrow_P^*(s)]$, hence the result. □

Proposition 4.5. *Given a program $\rightarrow_P$, a predicate on traces ψ, a set of controlled variables $\mathcal{A}_C$, and an oracle for trace property robust satisfaction $O^{\exists\forall}$, let ϕ be a predicate on memory state,*

(1)(a) *if $O^{\exists\forall}$ is correct for $\rightarrow_P, \mathcal{A}_C, \psi, \phi$ and $O^{\exists\forall}(\rightarrow_P, \mathcal{A}_C, \psi, \phi) = \top, a$ then ϕ is a sufficient robust trace property constraint for ψ in $\rightarrow_P$ with witness a,*

(b) *if $O^{\exists\forall}$ is complete for $\rightarrow_P, \mathcal{A}_C, \psi, \phi$ and ϕ is a sufficient robust trace property constraint for ψ in $\rightarrow_P$ with witness a then $O^{\exists\forall}(\rightarrow_P, \mathcal{A}_C, \psi, \phi) = \top, a$,*

(2)(a) *if $O^{\exists\exists}$ is complete for $\rightarrow_P, \psi, \neg\phi$ and $O^{\exists\exists}(\rightarrow_P, \psi, \neg\phi) = \bot, s$ then ϕ is a necessary trace property constraint for ψ in $\rightarrow_P$,*

(b) *if $O^{\exists\exists}$ is correct for $\rightarrow_P, \psi, \neg\phi$ and ϕ is a necessary trace property constraint for ψ in $\rightarrow_P$ then $O^{\exists\exists}(\rightarrow_P, \psi, \neg\phi) = \bot, s$.*

Definition 4.6 (weakest robust trace property constraint). Given a program $\rightarrow_P$, a predicate on traces ψ, a set of controlled variables $\mathcal{A}_C$, a predicate on memory states ϕ is a weakest robust trace property constraint if it is both a sufficient robust constraint and a necessary constraint.

4.2 Minimality Consideration

A point we have to consider *w.r.t.* the input inference language is that it may contain equivalent solutions and/or solutions that can be derived from another, more general solution. To prevent the generation of such solutions, we propose an ordering criterion that is similar to Echenim *et al.* [Echenim et al. 2018]. The general idea is to remove from our set of solutions the candidates that are entailed by another solution. Unfortunately, $\models$ is not a total order on formulas. This means that we need an additional criterion to deterministically choose between two equivalent solutions. This can be done with any (arbitrary) total, well-founded order $\prec$, for example the lexicographic order on the textual representation of the formula. Intuitively, this order should help prune candidates that are syntactically complex to obtain a more comprehensive result. For instance, while $a = 5 \wedge a \neq 6$ and $a = 5$ are logically equivalent, $\prec$ should permit to select the second predicate.

With these two consideration, we can define a notion of minimality according to the lexicographical order $(\models, \prec)$.

Definition 4.7 (minimal set of generating implicants). Given a set of predicates on memory states $\Phi = \{\phi_1, \ldots, \phi_n\}$, we denote by $\bigwedge_{min}(\Phi)$ the *minimal set of generating implicants of* Φ, that is the subset of Φ obtained by deleting from Φ any formula ϕ where there exists a formula $\phi' \in \Phi$ implied by ϕ ($\phi \models \phi'$) and either not equivalent to it ($\phi' \not\models \phi$) or smaller ($\phi' \prec \phi$).

This operator is used by the algorithm to iteratively prune redundant solutions (Lemma 4.8).

Lemma 4.8. *Let $\rightarrow_P$ be a program, ψ a trace property and $\mathcal{A}_C$ a set of controlled variables. Let $\Phi = \{\phi_1, \ldots, \phi_n\}$ be a set of predicates on memory states such that any $\phi_i \in \Phi$ is a sufficient robust trace property constraint for ψ in $\rightarrow_P$. We have that any $\phi \in \bigwedge_{min}(\Phi)$ is a sufficient robust trace property constraint for ψ in $\rightarrow_P$. Moreover, $\bigwedge_{min}(\Phi)$ does not contain any redundant sufficient robust trace property constraint: for any $\phi \in \bigwedge_{min}(\Phi)$, there are no $\phi' \in \bigwedge_{min}(\Phi), \phi' \neq \phi$ such that $\phi' \models \phi$.*

Proof. Let $\Phi = \{\phi_1, \ldots, \phi_n\}$ a set of predicates on memory states such that any $\phi_i \in \Phi$ is a sufficient robust trace property constraint for ψ in $\rightarrow_P$. The first point comes from $\bigwedge_{min}(\Phi)$ being a subset of Φ. For the second point, by contradiction assume we have $(\phi, \phi') \in \bigwedge_{min}(\Phi)$ such that $\phi \neq \phi'$ and $\phi' \models \phi$. Then by instantiating Definition 4.7 on ϕ', we get $\phi \models \phi' \wedge \neg(\phi \prec \phi')$. By instantiating Definition 4.7 again on ϕ, we get $\phi' \models \phi \wedge \neg(\phi' \prec \phi)$. As $\prec$ is antisymmetric, we deduce that $\phi = \phi'$, which contradicts our assumption. □

4.3 Baseline Algorithm

With these initial considerations, we can define an initial algorithmic framework for the generation of sufficient robust trace property constraints (Algorithm 1). It takes as inputs the inference language $\mathcal{G}$, the program $\rightarrow_P$, the target trace property ψ and the set of controlled variables $\mathcal{A}_C$. The algorithm returns a set of sufficient robust trace property constraints R.

Algorithm 1: BaselineRCInfer($\mathcal{G}, \rightarrow_P, \psi, \mathcal{A}_C$)

Input: $\mathcal{G}$: inference language, $\rightarrow_P$: program, ψ: target trace property, $\mathcal{A}_C$: controlled variables
Output: R: sufficient constraints
Parameters: $O^{\exists\exists}$: trace property oracle, $O^{\exists\forall}$: robust trace property oracle

```
1 if ⊤, s ← O^∃∃(→_P, ψ, ⊤) then                          // ensure ψ satisfiable
2     R ⟵ {y = s} if y = s ∈ G else ∅;                    // init R with satisfying trace
3     for φ ∈ G do
4         if O^∃∀(→_P, A_C, ψ, φ) then                    // check candidate φ
5             R ⟵ ⋀_min(R ∪ {φ});                         // update and minimize R
6             if ¬O^∃∃(→_P, ψ, ¬(⋁_{φ'∈R} φ')) then       // check weakest
7                 return R;
8     return R;
9 return {⊥};
```

This algorithm uses a classical validation loop over the constraints of $\mathcal{G}$, and the loop is adapted to verify robust trace properties. The generation and the verification of robust trace properties are handled by robust oracle queries. It is important to note that the algorithm makes no direct conclusion on the domain of the controlled variables for the generated constraints. Typically, when multiple sufficient robust trace property constraints are returned, they may not permit the robust satisfaction of ψ with the same controlled value.

While BaselineRCInfer exhibits good properties for the generation of robust trace property constraints (cf. Theorem 4.9), its naive exploration of the inference language (Line 3 of Algorithm 1) renders it prohibitively inefficient. In order to make the technique viable, we will propose in Section 5 strategies for the efficient exploration of $\mathcal{G}$, yielding an optimized inference algorithm (Algorithm 2).

Theorem 4.9. *Let $\rightarrow_P$ be a program, ψ a trace property, $\mathcal{A}_C$ a set of controlled variables and $\mathcal{G}$ an inference language.*

(1) *If both $O^{\exists\exists}$ and $O^{\exists\forall}$ terminate, then BaselineRCInfer **terminates**.*
(2) *If $O^{\exists\forall}$ is correct, then BaselineRCInfer is **correct**, that is, at any step of the algorithm, any $\phi \in R$ is a sufficient robust trace property constraint for ψ in $\rightarrow_P$.*
(3) *If $O^{\exists\exists}$ and $O^{\exists\forall}$ are complete for weakness oracle queries of Line 6, that is, for ψ and any assumption $\phi \in \mathcal{G} \cup \left\{\neg \bigvee_{\phi' \in \Theta} \phi' \;\middle|\; \Theta \subset \mathcal{G}\right\}$, then BaselineRCInfer is **complete** w.r.t. $\mathcal{G}$, that is, for any candidate $\phi \in \mathcal{G}$ that is a sufficient robust trace property constraint for ψ in $\rightarrow_P$, we have the existence of a witness $a \in \mathcal{S}_{\mathcal{A}_C}$ such that both $\exists s \in \mathcal{S}_{\mathcal{A}}, \phi[s] \wedge s|_{\mathcal{A}_C} = a$ and $\forall s \in \mathcal{S}_{\mathcal{A}}, (\phi[s] \wedge s|_{\mathcal{A}_C} = a) \Rightarrow \bigvee_{\phi' \in R} \phi'[s]$. Informally, this means that all the sufficient robust trace property constraint of $\mathcal{G}$ are encompassed by the ones returned by the algorithm.*
(4) *If $O^{\exists\exists}$ and $O^{\exists\forall}$ are correct and complete for all queries, that is, for ψ and any assumption $\phi \in \mathcal{G} \cup \left\{\neg \bigvee_{\phi' \in \Theta} \phi' \;\middle|\; \Theta \subset \mathcal{G}\right\}$, then*
   (a) *If BaselineRCInfer did not return from Line 7, then BaselineRCInfer is **minimal** w.r.t. $\mathcal{G}$, that is, for any candidate $\phi \in \mathcal{G}$ that is a sufficient robust trace property constraint for ψ in $\rightarrow_P$, either $\phi \in R$ or there exists $\phi' \in R$ such that $\phi \models \phi'$ and $\phi' \prec \phi$.*
   (b) *If BaselineRCInfer returned from Line 7, then $\bigvee_{\phi \in R} \phi$ is a **weakest** robust trace property constraint for ψ in $\rightarrow_P$.*

Proof. (1) This is obtained by terminating oracle queries and by construction as $\mathcal{G}$ is finite.
(2) R is only updated at Line 5 of Algorithm 1. This is only done under the condition that $O^{\exists\forall}(\rightarrow_P, \mathcal{A}_C, \psi, \phi) = \top, a$ (Line 4). By Definition 3.6, Definition 3.4 and Lemma 4.8, we have that R only contain sufficient robust trace property constraint after the update if this was verified initially. The initialization of R at Line 2 ensures that this is always the case.
(3) If we return at Line 8 or at Line 9 of Algorithm 1, completeness is ensured by Definition 3.7 and by the exhaustive exploration of $\mathcal{G}$. If we return at Line 7, by Proposition 4.5, we have that $\bigvee_{\phi \in R} \phi$ is a necessary trace property constraint for ψ in $\rightarrow_P$. We have the result by Proposition 4.4.
(4)(a) If we return at Line 9, this is a consequence of the absence of solutions in $\mathcal{G}$. If we return at Line 8, this is a consequence of the fact that all the candidates are tried at Line 4 and of Lemma 4.8.
   (b) This is a direct consequence of Proposition 2 and of the fact that disjunctions of sufficient robust trace property constraints are sufficient robust trace property constraints.

□

4.4 Discussion

Infinite inference language. When $\mathcal{G}$ contains an infinite number of candidates, the termination of BaselineRCInfer is preserved only if a weakest robust trace property constraint can be obtained by building disjunctions of elements of $\mathcal{G}$ and if the robust oracle $O^{\exists\forall}$ is complete for the corresponding query at Line 6 of Algorithm 1. Still, as Point 2 of Theorem 4.9 holds, it is possible to interrupt the algorithm at any stage to recover possible solutions to the abduction problem.

Initial assumption. When working on program verification, an initial restriction on input variables may already exist, either from external knowledge or to check a particular scenario. Algorithm 1 can be run under an initial assumption ϕ_0 to generate sufficient robust trace property constraints in this specific case. This can be done by replacing all assumptions of the oracle queries by their conjunction with ϕ_0. The results of Theorem 4.9 hold as long as we propagate the oracle correctness and completeness expectations for the conjunctions of assumptions with ϕ_0.

Timeouts. In practice, the verification time of some oracle queries can be unaffordable. For such cases, we can add a timeout on each query to reduce the execution time. This can be seen as a terminating under-approximation of the oracle, which means that the computation might miss some solutions. We can maintain, alongside the results of Algorithm 1, a set of candidate constraints for which the algorithm could not decide under the given timeout.

Undecidable inference languages. Constraints expressed in partially-decidable theories impose either incorrectness or incompleteness on the oracles used to verify them, which is again a case of over-approximating or under-approximating oracles. Alternatively, partial decidability can occur for entailment. This can arise at two locations, either in the pruning rules of Algorithm 1, in which case we can work as if the implication was not proven to ensure the completeness of the algorithm, or in the computation of $\bigwedge_{min}(\cdot)$. In this second case, we can also consider that the implication is not verified, which preserves the results of Theorem 4.9, except points 3 and 4.

5 STRATEGIES FOR EFFICIENT CONSTRAINTS ABDUCTION

The efficiency of the algorithmic framework presented in Section 4 thus strongly depends on our ability to reduce the number of candidates that need to be verified with an oracle query. Modern abduction algorithms have proposed clever strategies for inference language exploration [Echenim et al. 2018; Reynolds et al. 2020]. The core principle of such methods is the use of counter-examples to prune incorrect candidates, supplemented by information retrieval to cut redundant candidates and find satisfying solutions faster. In this Section, we build an exploration function by borrowing relevant candidate pruning strategies from those methods. We adapt the ideas for trace property and robust trace property verification (Section 5.2) and extend the framework to benefit from the joint generation of necessary robust trace property constraints (Section 5.1). Additionally, we study the construction of an exploration heuristic to guide the exploration of the constraint language (Section 5.3). The joint application of these strategies leads to the definition of a viable algorithmic framework for the generation of robust trace property constraints (Algorithm 2 and Algorithm 3). We show that the resulting framework preserves the results of Theorem 4.9 under the requirement of some additional scope for the oracles.

5.1 Reducing the Number of Robust Oracle Queries

First, it is possible to verify that at least a trace satisfying the target trace property exist under the current candidate constraint. When this verification fails, we can prevent an unnecessary robust oracle query. To do so, we precede robust oracle queries by a non-robust oracle check. The rationale is that we expect non-robustness verification to be both more likely to succeed—as this is required for robust satisfaction—and less costly than robust verification. When such tests fail, we can also derive a necessary constraint (Proposition 4.5). Additionally, we can keep track of a set of robust property-breaking constraints, denoted U, representing constraints that do not suffice to obtain robust trace property satisfaction.

Second, at each step, we can explicitly check and monitor whether a candidate is a necessary trace property constraint, which can, again, be kept in a set N. We can exploit such constraints to build a solution faster than the direct exploration of constraint language, by trying candidates built

by adding (conjunction) such constraints to the candidates of $\mathcal{G}$. However, this construction may create candidate constraints that are out of the inference language $\mathcal{G}$. To avoid this, we define an operator $\max_{\mathcal{G}}$ as follows: $\max_{\mathcal{G}} : \phi_{\mathcal{K}}, \mathcal{G}, N \mapsto \max_{\widehat{\preceq}}(\left\{N_{\phi_{\mathcal{K}}} \subset N \;\middle|\; \phi_{\mathcal{K}} \wedge \bigwedge_{\phi \in N_{\phi_{\mathcal{K}}}} \phi \in \mathcal{G}\right\})$ where $\widehat{\preceq} : N_1, N_2 \mapsto \textsc{Card}(N_1) < \textsc{Card}(N_2) \vee (\textsc{Card}(N_1) = \textsc{Card}(N_2) \wedge \bigwedge_{\phi \in N_1} \phi \prec \bigwedge_{\phi \in N_2} \phi)$. $\max_{\mathcal{G}}$ is used in the language exploration function (Algorithm 3).

Both necessary robust trace property constraints and robust property-breaking constraints can be reused to prune redundant candidates during the exploration of the inference language. We can prune candidates that are consequences of the known necessary trace property constraints, as such formulas are mechanically less restrictive necessary robust trace property constraints. A similar reasoning can be done for constraints breaking the non-robust satisfaction of the targeted trace property, as any constraint more restrictive only encompasses input states known not to produce a trace satisfying this trace property. Similarly, candidates that have for consequence a sufficient robust trace property constraint can be skipped as the corresponding solution is less general than what is already known.

5.2 Counter-Examples Guidance for Robust Constraints

Another usual inference language pruning strategy consists in the generation of consistent counter-examples against which candidate constraints can be checked for satisfiability. When all the variables are uncontrolled, a counter-example is a program input from which the target trace property cannot be satisfied. When a candidate constraint is satisfied by such an input state, we know that this candidate is not a sufficient robust trace property constraint as it is possible for an input state to both satisfy the constraint and not lead to the satisfaction of the target trace property.

However, in our case, when we consider a property that is not *robustly* satisfiable, we cannot obtain a counter-example with this method, as the uncontrolled variables are universally quantified. Moreover, the interleaving of controlled and uncontrolled variables can require complex counter-examples including case disjunctions on the sates of the controlled variables. When a candidate constraint ϕ is not a sufficient robust trace property constraint, we propose to look for a counter-example expressed as an uncontrolled memory state, compatible with ϕ, but from which the target property cannot be robustly satisfied. We propose to derive these counter-examples from more restrictive alternative robust satisfaction queries on an alternative property $\widehat{\psi}$ that ensures that ψ cannot be satisfied in the same trace. By testing the previously obtained counter-examples against yet-to-be-tested constraint candidates, we can skip those that are satisfied as they would not permit to achieve robust trace property satisfaction (we have already found a set of input values from which the target property is not satisfied).

Proposition 5.1. *Given a program $\rightarrow_P$, a set of controlled variables $\mathcal{A}_C$, a trace predicate ψ, let $\widehat{\psi}$ be a trace predicate such that $\forall t \in P^*, \widehat{\psi}[t] \Rightarrow \neg\psi[t]$.*

Assuming that $\widehat{\psi}$ is satisfiable, then for any $\mathcal{X} \subseteq \mathcal{A} \setminus \mathcal{A}_C$, if $\rightarrow_P$ robustly satisfies $\widehat{\psi}$ for $\mathcal{X}$ under ϕ (for, eg, a witness x), then any predicate on memory states ϕ' such that $\phi' \wedge y|_{\mathcal{X}} = x$ is satisfiable is not a sufficient robust constraint for ψ under ϕ.

A straightforward choice for the alternative set of controlled variables is $\mathcal{A}_{\overline{C}} = \mathcal{A} \setminus \mathcal{A}_C$. We point out that it is possible to use more than one such "non-satisfaction" trace property at the expense of more oracle tests when looking for counter-examples. Moreover, one can notice that this pruning strategy misses some of the more general cases for which no uncontrolled input would satisfy $\widehat{\psi}$ independently of the controlled input but it would still be possible to find a satisfying uncontrolled input for each controlled input.

5.3 Inference Language Ordering Heuristic

Finally, in order to help the algorithm to find a good candidate among the remaining ones, we propose a strategy for building an ordering heuristic on the elements of $\mathcal{G}$.

Reusing trace examples. As we are using additional non-robust oracle queries, we can maintain a set V of example traces that satisfy the target trace property. These traces can be used to sort the candidates by making the conjecture that the more of such traces satisfy a constraint, the more likely this constraint is to describe a set of inputs that can satisfy the target property. We propose for this a lexicographic sorting function using the sorting key: $\phi \mapsto (\mathit{count}(\wedge, \phi), -\textsc{Card}(\{s \in V \mid \phi[s]\}))$ on candidate constraints, where $\mathit{count}(\wedge, \phi)$ counts the number of conjunctions in ϕ.

Cutting from necessary robust trace property constraints. As this exploration heuristic works out of the presence of necessary robust trace property constraints, it is also possible to remove constraints that include their conjunction from the language, as long as we retry previously tested constraints each time the set of necessary robust trace property constraints is updated, to ensure completeness.

Improving the initial solution. Another thing that can be done is to use the initial witness example s obtained at Line 1 of Algorithm 2 to deduce that some variables are required to have the value they have in s for the trace to satisfy the target property. We can then initialize the set of necessary trace property constraints with the formula corresponding to these equalities. Such subsets can be found by enumeration. As a complete enumeration is too expensive, we propose to perform a partial enumeration on a set of user-given variables.

Note that such constraints may not be present in the inference language. If this is the case, we can either prevent unknown constraints from being added or allow the algorithm to return solutions that are conjunctions of an element of $\mathcal{G}$ with such constraints.

We will denote by *browse* the tool function we use to explore the inference language $\mathcal{G}$. When using the heuristics introduced in this section, it can be defined by $\mathit{browse} : \mathcal{G}, V \mapsto \mathit{sorted}(\mathcal{G}, \phi \mapsto \mathit{count}(\wedge, \phi), -\textsc{Card}(\{s \in V \mid \phi[s]\}))$ where V is a set of memory states from which we obtain the satisfaction of the target trace property. This is the definition we will use in all the following but note that other choices are possible as long as no candidates are missed.

5.4 Complete Algorithm for Abductive Inference

With these considerations, we propose a refined algorithm for the generation of sufficient robust trace property constraints, ARCInfer, presented in Algorithm 2. The algorithm takes as input an inference language $\mathcal{G}$, a program $\rightarrow_P$, a trace property ψ, a ψ-breaking trace property $\widehat{\psi}$ in the sense of Proposition 5.1, a set of controlled variables $\mathcal{A}_C$ and `prunef` = (`nec`, `cex`, `browse`), a tuple of boolean flags to decide whether we apply a given pruning strategy or not. It returns a triplet (R, N, U) where R is a set of sufficient robust trace property constraints, N is a set of necessary robust trace property constraints and U is a set of property-breaking constraints.

Algorithm 2 queries the exploration function NextRC (Algorithm 3) for selecting candidates from $\mathcal{G}$. For each iteration of the loop at Line 4 of Algorithm 2, a call to NextRC is made. Algorithm 3 is executed until it yields a candidate, at which point its execution is put on hold until the next time NextRC is queried. Algorithm 3 applies the pruning and ordering strategies presented in this section and maintains internally a set $\overline{V}$ of counter-examples to the robust satisfaction of the targeted trace property. The candidate is returned to Algorithm 2 as a tuple $\phi_{\mathcal{K}}, \phi, \delta_N, \delta_R$ where $\phi_{\mathcal{K}}$ is the initial candidate that was obtained from $\mathcal{G}$, ϕ is the actual candidate, built by adding the known necessary constraints and following the precautions mentioned in Section 5.1 and δ_N, δ_R are flags that indicate whether the candidate constraint ϕ should be tested as a necessary (resp.

Algorithm 2: ARCInfer($\mathcal{G}, \rightarrow_P, \psi, \widehat{\psi}, \mathcal{A}_C, \texttt{prunef}$)

Input: $\mathcal{G}$: inference language, $\rightarrow_P$: program, ψ: prop, $\widehat{\psi}$: prop breaking ψ, $\mathcal{A}_C$: controlled variables, prunef: strategy flags
Output: R: sufficient constraints, N: necessary constraints, U: breaking constraints
Note: $O^{\exists\exists}$: trace property oracle, $O^{\exists\forall}$: robust trace property oracle

1 **if** $\top, s \leftarrow O^{\exists\exists}(\rightarrow_P, \psi, \top)$ **then** // ensure ψ satisfiable
2 $V \longleftarrow \{s\}$; // init satisfying memory states examples
3 $R, N, U \longleftarrow \{y = s\}$ **if** $y = s \in \mathcal{G}$ **else** $\emptyset, \{\top\}, \{\bot\}$; // init result sets
4 **while** $\phi_{\mathcal{K}}, \phi, \delta_N, \delta_R \longleftarrow$ NextRC($\mathcal{G}, \rightarrow_P, \psi, \widehat{\psi}, \mathcal{A}_C, V, R, N, U, \texttt{prunef}$) **do** // explore $\mathcal{G}$
5 **if** δ_R ***and*** $\top, s \leftarrow O^{\exists\exists}(\rightarrow_P, \psi, \phi)$ **then** // ensure ψ satisfiable under ϕ
6 $V \longleftarrow V \cup \{s\}$; // new trace example
7 **if** $O^{\exists\forall}(\rightarrow_P, \mathcal{A}_C, \psi, \phi)$ **then** // check candidate ϕ
8 $R \longleftarrow \bigwedge_{min}(R \cup \{\phi\})$; // update and minimize R
9 **if** $\neg O^{\exists\exists}(\rightarrow_P, \psi, \neg(\bigvee_{\phi \in R} \phi))$ **then** // check weakest
10 **return** $(R, \{\bigvee_{\phi' \in R} \phi'\}, U)$;
11 **else**
12 $U \longleftarrow U \cup \{\phi\}$; // new breaking constraint
13 **else if** δ_R **then**
14 $N \longleftarrow N \cup \{\neg\phi\}$; // new necessary constraint
15 **if** δ_N ***and*** $\neg O^{\exists\exists}(\rightarrow_P, \psi, \neg\phi_{\mathcal{K}})$ **then**
16 $N \longleftarrow N \cup \{\phi_{\mathcal{K}}\}$; // new necessary constraint
17 **return** (R, N, U);
18 **return** $(\{\bot\}, \{\bot\}, \{\bot\})$;

sufficient) constraint in Algorithm 2. Note that we chose to always check if candidates are necessary constraints as long as the option to build them is set. Look for better heuristics could help mitigate the number additional oracle queries that these checks produce while still finding enough necessary constraints to permit the algorithm to find complex constraints.

Theorem 5.2. *Let $\rightarrow_P$ be a program, ψ a trace property, $\widehat{\psi}$ a trace property breaking ψ, $\mathcal{A}_C$ a set of controlled variables, $\mathcal{G}$ an inference language and* $\texttt{prunef} = (\texttt{nec}, \texttt{cex}, \texttt{browse})$ *a tuple of boolean flags.*

(1) *If* $\texttt{prunef} = (\bot, \bot, \bot)$ *then all points of Theorem 4.9 hold for ARCInfer.*
(2) *If* $\texttt{prunef.browse} = \top$ *then all points of Theorem 4.9 are preserved.*
(3) *If* $\texttt{prunef.cex} = \top$ *and $O^{\exists\exists}$ and $O^{\exists\forall}$ are correct for $\widehat{\psi}$ with any assumption in $\mathcal{G}$, then all points of Theorem 4.9 are preserved.*
(4) *If* $\texttt{prunef.nec} = \top$ *and $O^{\exists\exists}$ and $O^{\exists\forall}$ are complete for ψ and any assumption of $\mathcal{G}$ and their negation, then Point 1, Point 2, Point 3 and Point 4b are preserved. Point 4a is not preserved as the necessary constraints are added greedily to the candidate constructed at Line 3 of Algorithm 3–the resulting constraint is thus not necessarily minimal* w.r.t. $\prec$. *ARCInfer still remains* ***minimal-equivalent*** w.r.t. $\mathcal{G}$, *that is, for any candidate $\psi \in \mathcal{G}$ that is a sufficient robust trace property constraint for ψ in $\rightarrow_P$, either $\psi \in R$ or there exists $\phi \in R$ such that ψ is equivalent to ϕ.*

Algorithm 3: NextRC($\mathcal{G}, \rightarrow_P, \psi, \widehat{\psi}, \mathcal{A}_C, V, R, N, U,$ prunef)

Input: $\mathcal{G}$: inference language, $\rightarrow_P$: program, ψ: prop, $\widehat{\psi}$: prop breaking ψ, $\mathcal{A}_C$: controlled variables, V: examples of input states of $\rightarrow_P$ satisfying ψ, R: known sufficient constraints, N: known necessary constraints, U: known breaking constraints, prunef: strategy flags

Output: $\phi_{\mathcal{K}}$: core candidate, ϕ: candidate, δ_N: check for necessary flag, δ_R: check for sufficient flag

Note: $O^{\exists\exists}$: oracle for trace property satisfaction, $O^{\exists\forall}$: oracle for robust trace property satisfaction

1 $\overline{V} \longleftarrow \emptyset$; // init. counter-examples
2 **for** $\phi_{\mathcal{K}} \in browse(\mathcal{G}, V)$ **if** prunef.browse **else** $\mathcal{G}$ **do** // get candidate from $\mathcal{G}$
3 $\quad \phi \longleftarrow \phi_{\mathcal{K}} \wedge \bigwedge_{\phi' \in \max_{\mathcal{G}}(\phi_{\mathcal{K}}, \mathcal{G}, N)} \phi'$ **if** prunef.nec **else** $\phi_{\mathcal{K}}$; // add nec. constraints
4 $\quad$ **if** ϕ *is unsatifiable* **then**
5 $\quad\quad$ **continue**; // skip: inconsistent
6 $\quad$ **if** prunef.cex **and** $\exists m, \mathcal{X} \in \overline{V}, \phi \wedge y|_{\mathcal{X}} = m$ *is satisfiable* **then**
7 $\quad\quad$ **continue**; // skip: sat. by counter-example
8 $\quad$ **if** $\exists \phi_s \in R, \phi \models \phi_s$ **then**
9 $\quad\quad$ **continue**; // skip: stronger than known suff. constraint
10 $\quad$ **if** prunef.nec **and** $\exists \phi_u \in U, \phi_u \models \phi$ **then**
11 $\quad\quad$ **continue**; // skip: weaker than known break. constraint
12 $\quad$ **if** prunef.nec **and** $(\bigwedge_{\phi_n \in N} \phi_n) \models \phi$ **then**
13 $\quad\quad$ **continue**; // skip: weaker than known nec. constraint
14 $\quad$ **if** prunef.cex **and** $\top, cex \longleftarrow O^{\exists\forall}(\rightarrow_P, \mathcal{X}, \widehat{\psi}, \phi)$ *for* $\mathcal{X} \subseteq \mathcal{A} \setminus \mathcal{A}_C$ **then**
15 $\quad\quad \overline{V} \longleftarrow \overline{V} \cup \{cex\}, \mathcal{X}$; // new counter-example
16 $\quad\quad$ **yield** $\phi_{\mathcal{K}}, \phi,$ prunef.nec, $\bot$; // forward for nec. check
17 $\quad$ **else**
18 $\quad\quad$ **yield** $\phi_{\mathcal{K}}, \phi,$ prunef.nec, $\top$; // forward for nec. and suff. checks

Moreover,

(5) *If $O^{\exists\exists}$ is complete for ψ and any assumption of $\mathcal{G}$ and their negation, then ARCInfer is **correct for necessary constraints**, that is, at any step of the algorithm, any $\phi \in N$ is a necessary trace property constraint for ψ in $\rightarrow_P$.*

(6) *If $O^{\exists\forall}$ is complete for ψ and any assumption $\phi \in \mathcal{G}$, then ARCInfer is **correct for breaking constraints**, that is, at any step of the algorithm, for any $\phi \in U$, ϕ is not a robust trace proerty constraint for ψ.*

Proof. (1) Termination is ensured as we still only make oracle queries. Correctness is ensured as R is not updated at any other point. Completeness and minimality are ensured as we only prune candidates from Line 4 of Algorithm 3, which ensures that Definition 3.4 does not hold, and from Line 8 of Algorithm 3, where the absence of pruning will lead to the removal of the candidate from R by the application of $\bigwedge_{min}(\cdot)$.

(2) No element of $\mathcal{G}$ is missed, they are tested in a different order.

(3) This is a consequence of Proposition 1 and Proposition 5.1.

(4) Point 5 and Point 6 ensure that the constraints of N and U are correctly constructed. By Proposition 4.4, we have that only non sufficient robust constraints conditions are pruned.
(5) This a direct consequence of Proposition 2.
(6) This a direct consequence of Proposition 1.

□

5.5 Discussion

All the points discussed in Section 4.4 remain applicable to ARCInfer. We mentioned two additional points the relevance of which is induced by our exploration strategies.

Candidates constraints on controlled variables. While it is possible to populate the inference language with candidate constraints involving the controlled variables, this is likely to interfere with the robust oracle queries. Indeed, by computing concrete values for controlled variables, the robust oracle inherently builds stronger constraints. This means that such constraints are useless for ensuring robust satisfiability. While additional initial constraints on controlled variables may be added for problem definition, it is thus useless to include them in the inference language.

A trickier case arises for constraints constraining both controlled and uncontrolled variables. While, again, there are no correctness issues, the details are harder. This means that if it is necessary to link controlled variables to uncontrolled variables to obtain a sufficient constraint, it is unlikely to be guided by the ordering heuristic. Moreover, as part of the variables are more strongly constrained, they are also unlikely to be efficiently pruned from a counter-example.

Inference language construction. ARCInfer is well adapted for inference languages that can be constructed as conjunctions of literals for which both the positive and negative variants can be considered. This type of construction ensures that all the assumptions of the queries during the execution of ARCInfer remain in the inference language, which simplifies the oracles requirements. Moreover, this construction mingles well with the reuse of the necessary robust trace property constraints due to the constraint construction at Line 3 of Algorithm 3.

General abductive generation. If we replace oracle queries with SMT queries for property satisfaction, Algorithm 1 falls back to an abduction algorithm for first-order predicates. In particular, this means that the language-exploration strategies exposed in Section 5 could be applied to first-order abduction problems for $\exists\forall$ formulas. More generally, the framework presented in this paper is not inherently limited to the abductive inference of program trace properties. As long as oracles for verifying a given contextualized predicate and its robust counterpart can be defined, the same methodology can be applied without adaptation. Indeed, neither the algorithmic framework nor the exploration strategies of Section 5 use the underlying program out of the oracle queries.

6 EXPERIMENTAL EVALUATION

We now present an evaluation of our abduction algorithm. We build this evaluation on existing oracles for trace property and robust trace property satisfiability: location reachability and location robust reachability properties (Section 6.1). We detail our implementation of the abduction algorithm (Section 6.2), we evaluate our approach on two benchmarks built from sv-comp [Beyer 2012] and fissc [Dureuil et al. 2016] (Section 6.3), and we assess the evaluation results *w.r.t.* the following research questions (Section 6.4):

RQ1. Can ARCInfer compute non-trivial robust k-reachability constraints in non-robust cases?
RQ2. Can ARCInfer compute weakest robust k-reachability constraints (in the sense of Definition 4.3)?
RQ3. What are the algorithmic performances of ARCInfer for language exploration?

RQ4. How do the strategies of Section 5 influence the performance of ARCInfer?

We additionally propose a detailed applicative case study for the vulnerability assessment of the VerifyPIN benchmarks (fissc) against fault injection attacks (Section 6.5).

6.1 Restriction to k-Reachability

For practical reasons, we restrict our evaluation to the search of sufficient robust constraints for k-reachability properties. As mentioned in Section 3.4, k-reachability properties are trace properties that decide whether a given trace executes a program instruction at a selected target location ℓ, after at most k instructions (*i.e.* k consecutive applications of $\rightarrow_P$ from the initial memory state). Oracles for k-reachability and robust k-reachability under assumption include symbolic execution and robust symbolic execution. Correct and complete algorithms for k-bounded symbolic execution and k-bounded robust symbolic execution are available in the literature [Cadar and Sen 2013; Girol et al. 2021]. These algorithms generate *path predicates*, that are first-order formulas representing the execution of a given program path. They enumerate all the paths of at most k successive instructions and use an underlying SMT-Solver [Barbosa et al. 2022; Barrett et al. 2009; de Moura and Bjørner 2008] to look for the existence of a witness.

k-reachability breaking properties. In order to apply the counter-example guidance described in Section 5.2, we need to define properties that ensure the non-k-reachability of the target location. This can be done by a k-reachability property to an alternative target location $\overline{\ell}$ in a concurrent path *w.r.t.* ℓ if ℓ is outside of an unbounded loop. Otherwise, this can be done by merging the reachable locations after exactly k instructions, and by instrumenting the program to jump at an alternative location after the instruction at location ℓ is executed. Note that this second case is not applicable for unbounded reachability.

Oracle definitions. We define $O_{\mathrm{SE}}^{\exists\exists}$ and $O_{\mathrm{SE}}^{\exists\forall}$ our oracles for bounded reachability and robust bounded reachability as calls to the aforementioned symbolic execution and robust symbolic execution algorithms. The witness can be obtained from the model returned by the SMT-Solver when a solution is found.

6.2 Implementation and Setup

PyAbd is a Python program implementing ARCInfer (Algorithm 2) and function NextRC (Algorithm 3) for k-reachability (Section 6.1). It leverages Binsec [David et al. 2016; Djoudi and Bardin 2015] and Binsec/RSE [Girol et al. 2021, 2022], for the symbolic execution and robust symbolic execution queries respectively, and the SMT solver cvc5 [Barbosa et al. 2022] for pruning candidate constraints on inconsistency, counter-example satisfiability and for checking logical implication between constraints. Since robust symbolic execution depends on quantified formulas, we use Z3 [de Moura and Bjørner 2008] for all the (robust) symbolic execution queries. We consider the theory of bitvectors and arrays (QF_ABV), well adapted for binary-level code analysis.

Inputs. PyAbd takes as main inputs a binary file under analysis (our experiments span the x86 and ARMv7-M architectures), specifications of reachability targets (target property and for counter-example guidance), and a file describing the inference language.

Additional parameters. PyAbd supports the separate activation of each pruning and ordering rules in Algorithm 3, in order to evaluate their impact. In the following, we denote by PyAbd or `pyabd+allopt` the execution of Algorithm 3 with `prunef` $= (\top, \top, \top)$, by `pyabd+cex` counter-examples pruning only (`prunef` $= (\bot, \top, \bot)$), by `pyabd+nec` necessary constraints pruning only (`prunef` $= (\top, \bot, \bot)$ and the initialization in Section 5.3), and by `pyabd+noopt` the use of no

$$
\begin{array}{l}
\langle constraint \rangle ::= \langle constraint \rangle \wedge \langle constraint \rangle \\
\quad \mid \neg \langle nliteral \rangle \\
\quad \mid \langle literal \rangle \\
\\
\langle literal \rangle ::= \langle nliteral \rangle \\
\quad \mid *a_{32} = v_{32}
\end{array}
\qquad
\begin{array}{l}
\langle nliteral \rangle ::= *a_8 = *a'_8 \\
\quad \mid *a_{32} = *a'_{32} \\
\quad \mid *a_{32} = \texttt{0x000000:}(*a_8) \\
\quad \mid *a_8 = v_8 \\
\quad \mid *a_{32} = \texttt{0x000000:}v_8
\end{array}
$$

Fig. 3. Template grammar for the $E_{\mathcal{G}}(\Sigma_{\mathcal{A}_8}, \Sigma_{\mathcal{A}_{32}}, \Sigma_{\mathcal{V}_8}, \Sigma_{\mathcal{V}_{32}})$ inference language; depends on the initial sets of addresses mapping to 8-bit values ($\Sigma_{\mathcal{A}_8}$, with $a_8 \in \Sigma_{\mathcal{A}_8}$), addresses mapping to 32-bit values ($\Sigma_{\mathcal{A}_{32}}$, with $a_{32} \in \Sigma_{\mathcal{A}_{32}}$), 8-bit values ($\Sigma_{\mathcal{V}_8}$, with $v_8 \in \Sigma_{\mathcal{V}_8}$) and 32-bit values ($\Sigma_{\mathcal{V}_{32}}$, with $v_{32} \in \Sigma_{\mathcal{V}_{32}}$), and where $\texttt{0x000000:}v_8$ denotes the zero-extension of a 8-bit value to a 32-bit value

$$
\begin{array}{l}
\langle nliteral \rangle ::= *a_8 \leq *a'_8 \\
\quad \mid *a_{32} \leq *a'_{32} \\
\quad \mid *a_8 \leq v_8
\end{array}
$$

Fig. 4. Template grammar of the additional rules for the $I_{\mathcal{G}}(\Sigma_{\mathcal{A}_8}, \Sigma_{\mathcal{A}_{32}}, \Sigma_{\mathcal{V}_8}, \Sigma_{\mathcal{V}_{32}})$ inference language; depends on the initial sets of addresses mapping to 8-bit values ($\Sigma_{\mathcal{A}_8}$, with $a_8 \in \Sigma_{\mathcal{A}_8}$), addresses mapping to 32-bit values ($\Sigma_{\mathcal{A}_{32}}$, with $a_{32} \in \Sigma_{\mathcal{A}_{32}}$), 8-bit values ($\Sigma_{\mathcal{V}_8}$, with $v_8 \in \Sigma_{\mathcal{V}_8}$) and 32-bit values ($\Sigma_{\mathcal{V}_{32}}$, with $v_{32} \in \Sigma_{\mathcal{V}_{32}}$

additional strategies ($\texttt{prunef} = (\bot, \bot, \bot)$), which is equivalent to running BaselineRCInfer as long the inference language does not contain unsatisfiable constraints.

6.3 Benchmarks

SV-COMP. We automatically adapt C verification tasks from the sv-comp benchmarks [Beyer 2012] to binary-level reachability problems (x86 architecture). We select programs from the categories `bitvector`, `list`, `loops`, `nla-digbench`, `ntdrivers-simplified`, and `openssl-simplified`. The program sizes range from 30 to 1752 loc, with an average of 115 loc. Out of 147 programs, we select the 64 ones for which Binsec reaches the target location in less than 60 seconds and Binsec/RSE terminates under 60 seconds.

FISSC. fissc [Dureuil et al. 2016] is a collection of C programs with counter-measures against fault injection attacks [Yuce et al. 2018]. We select the VerifyPIN benchmarks, which present 10 variants of a PIN authentication procedure. The programs are compiled for the ARMv7-M architecture, and we exhaustively simulate, on each program, the skipping of a single machine instruction using code mutation, leading to 4810 distinct mutant binary programs. We select the 719 mutants for which Binsec reaches the target location (defined as an authentication with distinct input and secret PINs) under 60 seconds. Binsec/RSE also terminates under 60 seconds for all these programs.

Inference languages. Consider an initial set of addresses mapping to 8-bit values $\Sigma_{\mathcal{A}_8}$, an initial set of addresses mapping to 32-bit values $\Sigma_{\mathcal{A}_{32}}$, an initial set of 8-bit values $\Sigma_{\mathcal{V}_8}$ and an initial set of 32-bit values $\Sigma_{\mathcal{V}_{32}}$, which contain selected memory addresses and values for the program we consider. A first language, denoted $E_{\mathcal{G}}(\Sigma_{\mathcal{A}_8}, \Sigma_{\mathcal{A}_{32}}, \Sigma_{\mathcal{V}_8}, \Sigma_{\mathcal{V}_{32}})$, or simply $(E_{\mathcal{G}})$, is constructed from the conjunction of equalities and disqualities between the aforementioned memory addresses and values. This language is constructed from the grammar presented in Figure 3. We curate this language further to exclude the most trivial semantically equivalent constraints.

A second language, denoted $I_{\mathcal{G}}(\Sigma_{\mathcal{A}_8}, \Sigma_{\mathcal{A}_{32}}, \Sigma_{\mathcal{V}_8}, \Sigma_{\mathcal{V}_{32}})$, or simply $(I_{\mathcal{G}})$, adds inequalities between selected memory addresses and selected values. It is constructed by adding the rules of the grammar presented in Figure 4 to the grammar of Figure 3. Again, $I_{\mathcal{G}}(\Sigma_{\mathcal{A}_8}, \Sigma_{\mathcal{A}_{32}}, \Sigma_{\mathcal{V}_8}, \Sigma_{\mathcal{V}_{32}})$ is curated to exclude semantically equivalent elements.

A detailed description of how we build the sets $\Sigma_{\mathcal{A}_8}, \Sigma_{\mathcal{A}_{32}}, \Sigma_{\mathcal{V}_8}, \Sigma_{\mathcal{V}_{32}}$ for each target program is available in Appendix B.

Finally, note that each set can be extended by user-provided elements. In particular, when we use controlled addresses, we add the controlled addresses to $\Sigma_{\mathcal{A}_8}$ (or $\Sigma_{\mathcal{A}_{32}}$ if we control registers).

Controlled variables. We consider two configurations:

- ■ a configuration with controlled variables, denoted ■, specific to each benchmark, to evaluate PyAbd for the generation of sufficient robust reachability constraints;
- □ a configuration with no controlled variables, denoted □, to evaluate PyAbd for the generation of sufficient reachability constraints.

Note that in □ the method is equivalent to a standard precondition generator. Regarding controlled variables (■): for sv-comp they are arbitrarily chosen among the program variables, while for FISSC we define the input PIN bytes as controlled and the rest of the memory addresses as uncontrolled. This configuration supports the authentication with an incorrect input PIN without prior knowledge.

Experimental Setup. The global timeout for a PyAbd run is set to 1 hour per input program. The timeouts for Binsec and Binsec/RSE is set to 60 seconds in all contexts. All the SMT solver calls have a timeout of 10 seconds. All our tests are run in parallel on an Intel(R) Xeon(R) Gold 5220 CPU @2.20GHz machine with 36 cores, 72 threads, and 96 GB of RAM.

6.4 Results

RQ1—Sufficient robust k-reachability constraints. Table 1 shows the number of programs for which PyAbd finds a robust k-reachability constraint distinct from the one returned by its initial symbolic execution query (Line 3 of Algorithm 2), either with controlled variables (■) or without (□). We distinguish between sufficient robust k-reachability constraints only (# of sufficient rrc) and robust k-reachability constraints that are also weakest k-reachability constraints (# of weakest rrc). The table also contains, for comparison purposes, the number of programs for which the target location is robustly k-reachable (# of robust cases).

PyAbd always characterizes more programs than Binsec/RSE. With controlled variables (■), for sv-comp, 75 % (111/147) of the programs are inherently robustly k-reachable, but PyAbd finds a sufficient characterization for almost half of the remaining cases. For fissc, PyAbd generates a sufficient robust k-reachability constraint for half of the programs. Without controlled variables (□), PyAbd generates constraints for significantly more programs, because such constraints exhibit a link between the previously controlled and uncontrolled variables, and such constraints cannot be computed when those variables are separated in the robust reachability setup.

For sv-comp, adding inequalities to the inference language permits to find a solution in more cases. This is because such programs require inequality constraints on inputs to ensure robust k-reachability. The opposite happens for fissc, as equality and disequality suffice to express satisfying solutions. As the inference language is larger, PyAbd sometimes fail to reach the solution it found with the smaller $(E_{\mathcal{G}})$ language under the timeout. Still, this concerns at most 9 programs, in the setup without controlled variables, which shows that our approach scales relatively well *w.r.t.* the size of its inference language.

Conclusion. These results show that our method to compute robust k-reachability constraints can find non-trivial sufficient constraints, refining the results from robust symbolic execution.

Table 1. PyAbd analysis results for each benchmark configuration, with (■) or without (□) controlled variables. We detail the total number of programs analyzed, the number of programs for which the target location is robustly k-reachable, and the number of programs for which PyAbd finds sufficient robust k-reachability constraints (rrc), split between any sufficient rrc, and the weakest rrc constraints only.

	sv-comp ($E_{\mathcal{G}}$)		sv-comp ($I_{\mathcal{G}}$)		fissc ($E_{\mathcal{G}}$)		fissc ($I_{\mathcal{G}}$)	
	■	□	■	□	■	□	■	□
# programs	147	64	147	64	719	719	719	719
# of robust cases	111	3	111	3	129	118	129	118
# of sufficient rrc	122	5	127	24	359	598	351	589
# of weakest rrc	111	3	120	4	262	526	261	518

RQ2—Weakest robust k-reachability constraints. We look again at Table 1 to characterize if the generated robust k-reachability constraints (rrc) describe the complete input set that k-reaches a target location (weakness). For sv-comp, no weakest rrc is found with ($E_{\mathcal{G}}$) for cases that were not inherently robustly k-reachable. With the larger language ($I_{\mathcal{G}}$), PyAbd generates 9 and 1 additional complete robust k-reachability characterizations, resp. with or without controlled variables. For fissc, with controlled variables, PyAbd doubles the number of cases for which we find a weakest robust k-reachability constraint *w.r.t.* Binsec/RSE. Without controlled variables, most of the constraints found by PyAbd are weakest, suggesting that, when the inference language is expressive enough, PyAbd is able to generate a complete characterization. Moreover, the use of larger than necessary inference language ($I_{\mathcal{G}}$) does not degrade significantly the number of weakest constraint found (1 less with controlled variables, 8 without).

Conclusion. PyAbd is indeed able to infer weakest constraints of robust reachability when the underlying inference language is expressive enough.

RQ3—Inference language exploration. Table 2 details a number of metrics monitored during the execution of PyAbd on each benchmark, on three distinct subsets of programs: any PyAbd result (*), PyAbd finds a sufficient constraint ($\top$) and PyAbd finds a weakest constraint, including robust programs (W). The metrics reported are averages for the number of literals in $\mathcal{G}$, the number of candidates that are considered by PyAbd, the number of candidates that are evaluated by a robust oracle query, the number of returned solutions and the length of solutions.

Note (1) that the average number of considered candidates is increased by the programs for which no solution is found, as the whole inference language has to be explored, and (2) that the average number of literals is decreased by the robust cases, for which it is 0 as no inference language generation has been made by PyAbd.

The number of tested candidates before the weakest robust k-reachability constraint is found is 2.3 for the fissc benchmark with controlled variables with the ($E_{\mathcal{G}}$) inference language (resp. 2.2 with the ($I_{\mathcal{G}}$) inference language), 6.8 without controlled variables (resp. 11.7). For sv-comp, with the ($I_{\mathcal{G}}$) language, these numbers are 1.7 with controlled variables and 4.7 without. This shows that, with the help of necessary constraints and the browse heuristic, our method is much more efficient, being able to converge to the complete solution by trying a small number of candidates.

The average length of the generated constraints remains relatively small when PyAbd finds a weakest robust k-reachability constraint. If we remove the cases where the target location was already robustly k-reachable (present in Table 2), this gives 1.7 and 3.1 conjunctive elements for fissc respectively with and without controlled variables with the ($E_{\mathcal{G}}$), 1.8 and 1 for sv-comp with the ($I_{\mathcal{G}}$) language. When the weakest robust k-reachability constraint could not be computed, both the number of solutions and their complexity tend to increase, up to an average 5.8 constraints

Table 2. Internal metrics monitored during the execution of PyAbd, for each benchmark configuration, with (■) or without (□) controlled variables. Each line reports, for each configuration, the average number ($\hat{\#}$): of literals in the inference language, of candidates considered, of candidates checked, of returned robustly k-reachable constraints, of language literals in the returned constraints. The three groups of lines report, from top to bottom, metrics for: all the cases including timeouts (*), cases for which PyAbd finds a sufficient constraint (⊤), and cases for which PyAbd finds a weakest constraint (W).

		sv-comp ($E_\mathcal{G}$)		sv-comp ($I_\mathcal{G}$)		fissc ($E_\mathcal{G}$)		fissc ($I_\mathcal{G}$)	
		■	□	■	□	■	□	■	□
(*)	$\hat{\#}$ literals in $\mathcal{G}$	560.0	3463.7	862.2	5994.4	771.7	2141.6	1125.3	3233.3
	$\hat{\#}$ C. considered	63334.0	78665.5	13175.4	24457.9	11646.0	14032.3	18259.5	13905.4
	$\hat{\#}$ C. checked	51.0	577.3	37.1	583.4	121.2	126.6	117.7	127.0
	$\hat{\#}$ solutions	5.8	1.6	1.4	1.0	1.5	0.9	1.4	0.9
	avg. solution length	0.3	0.1	0.3	0.5	0.6	1.9	0.5	1.8
(⊤)	$\hat{\#}$ literals in $\mathcal{G}$	78.1	247.6	224.8	3175.6	436.6	1638.4	587.8	2589.5
	$\hat{\#}$ C. considered	10372.8	43418.0	7246.1	26265.3	7770.7	5747.2	11766.6	5099.3
	$\hat{\#}$ C. checked	32.5	366.4	15.5	760.2	77.4	70.3	74.3	74.2
	$\hat{\#}$ solutions	7.0	21	1.7	2.8	2.9	1.0	2.7	1.1
	avg. solution length	0.4	0.8	0.4	1.2	1.1	2.3	1.0	2.2
(W)	$\hat{\#}$ literals in $\mathcal{G}$	–	–	108.3	119.0	278.8	662.4	385.2	1156.1
	$\hat{\#}$ C. considered	1.0	1	34.3	7.2	41.0	205.3	61.0	41.0
	$\hat{\#}$ C. checked	1.0	1	1.7	5	2.3	6.8	2.2	11.7
	$\hat{\#}$ solutions	1.0	1	1.3	1.8	1.1	1.0	1.1	1.0
	avg. solution length	0.2	0	0.2	0.2	0.9	2.4	0.8	2.2

with an average of 0.3 conjunctive elements for the sv-comp benchmark with controlled variables on the ($I_\mathcal{G}$) language. This is typical of inference languages that are not expressive enough, and on which we can only iterate over a set of incomplete solutions that describe existing subsets of a weakest robust k-reachability constraint.

Conclusion. Our method explores the inference language much more efficiently than the baseline algorithm. In particular, a very small proportion of candidates are actually tested with a robust symbolic execution query. Moreover, when weakest constraints are generated, the total number of considered candidates is small. Otherwise, sufficient constraints are found while only exploring a small portion of the inference language.

RQ4—Influence of the strategies of Section 5. We look at the influence of optimization strategies (Section 5). Figure 5 shows the impact of different choices for the `prunef` parameter of Algorithm 2 on the generation of the first sufficient robust k-reachability constraint, on sv-comp with ($I_\mathcal{G}$) (top), and on fissc with ($E_\mathcal{G}$) (right), both with (left) and without (right) controlled variables. More detailed plots are available in the supplementary material.

Overall, the combination of all the strategies in PyAbd is better than any other tested combination. This means that each strategy was observed to reduce analysis time. It can be noted that when we use necessary k-reachability constraints (`prunef.nec` = ⊤), this may degrade the performance on a few cases because of the additional computation time required at initialization (sv-comp, □). In all the other cases, each strategy improves the capabilities of PyAbd to generate sufficient robust k-reachability constraints.

By detailing internal metrics for when all strategies are used, we observed that each strategy influences the algorithm in a very different way. The use of counter-examples is where most of the

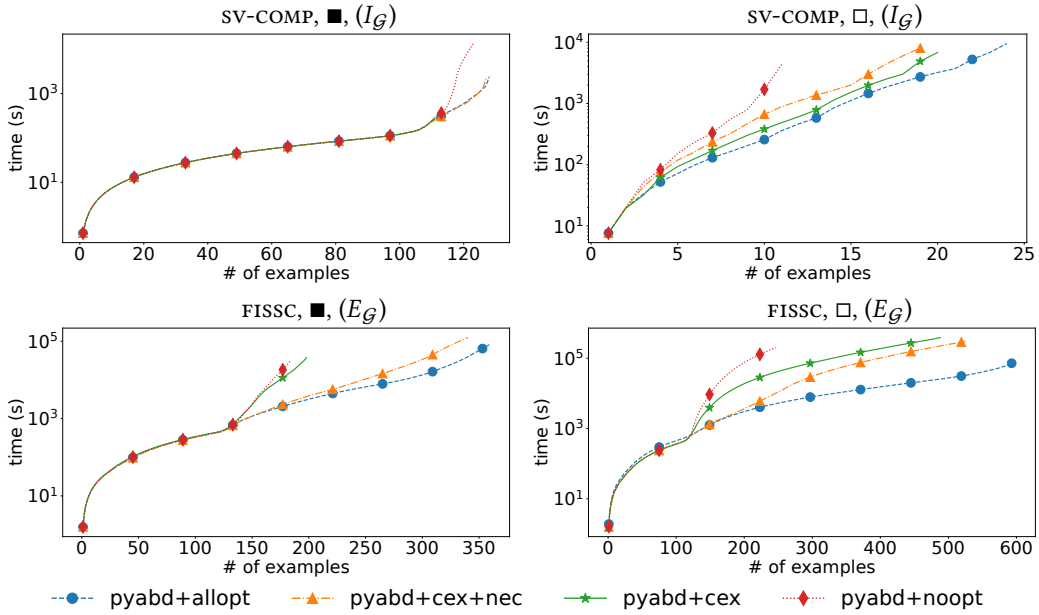

Fig. 5. Cactus plot showing the influence of the strategies of Section 5 on the computation of the first sufficient k-reachability constraint with PyAbd.

pruning comes from. Conversely, necessary constraints do not help much with pruning candidates but permit to populate candidate solutions with required elements quickly, which helps finding a solution faster as we do not have to wait to reach this element during the inference language exploration. Finally, the use of the ordering heuristic permits to try more relevant candidates at the beginning, which tends to help with the global language exploration.

Conclusion. All the strategies of Section 5 improve the efficiency of PyAbd, *w.r.t.* number of programs for which we find a solution under a time constraint, and computation time for each solution.

General conclusions. While PyAbd struggles with the exploration of the inference language for the baseline case, the use of the strategies of Section 5 permit the viable generation of sufficient robust k-reachability constraints, among which many are weakest constraints when the inference language is expressive enough. Also, the method scales relatively well *w.r.t.* to the size of the inference language, and each strategy of Section 5 has its use. Finally, these strategies permit to significantly reduce the number of oracle queries to make in order to obtain a satisfying solution.

6.5 Case Study: Vulnerability Assessment for FIA on the FISSC VerifyPIN

We now consider an applicative case study based on the analysis of the VerifyPIN benchmarks (FISSC). The underlying security scenario is the following: a security expert uses software-based fault injection methods to drive a fault injection attacks campaign on expensive hardware benches [Hoffmann et al. 2021]. In this setting, it is important to find as many vulnerabilities as possible, but also to avoid reporting unrealistic vulnerabilities, as they would waste both expert and equipment time.

Table 3 (top group of rows) reports the results of several vulnerability analysis methods, including the use of ISA-level simulation (Qemu) which is typical of this case study. In the FISSC benchmarks,

simulation using arbitrary inputs (Qemu) discovers 169 *exploitable* fault locations (real vulnerabilities), while symbolic execution (binsec) reports 719 *potential* vulnerabilities. Robust symbolic execution (Binsec/RSE) confirms 118 robustly reachable vulnerabilities (the most serious), which were all detected by the simulation, but does not help concluding for the many other potential cases found by the symbolic execution. Our method characterizes 598 of the potential vulnerabilities reported by binsec, where the returned constraint helps classifying the risk associated with each fault injection location. Typically, the constraints generated for 226 cases show that an unauthorized authentication under fault injection is possible with a reduced number of known PIN digits. Conversely, vulnerabilities that require to initialize some registers with exact initial values are usually discarded in a threat model excluding write access to the target memory.

To improve the security assessment of each vulnerability, a security expert typically evaluates the expected number of vulnerable inputs for each vulnerability. In the fissc benchmark, this can be counted as the minimal percentage of tuple values (user PIN, card PIN) for which unauthorized authentication is possible, reported in Table 3 (bottom group of rows). The analysis with PyAbd is based on ($E_{\mathcal{G}}$); the evaluation of the expected number of vulnerable inputs is based on an enumeration of the generated constraints, and we distinguish between two assessment methods: PyAbd$^{\mathrm{O}}$ (optimistic) considers that subconstraints containing input variables other than PIN are always evaluated to false, and PyAbd$^{\mathrm{P}}$ (pessimistic) considers that subconstraints containing such variables are always evaluated to true. Typically, in a robust constraint such as userPIN[0] == R0, as the register R0 is not a PIN variable, the constraint is always evaluated to false for $PyAbd^{O}$ and to true for $PyAbd^{P}$.

We also consider two simulation-based analysis methods: Qemu emulates program execution with a single, arbitrarily chosen, set of program inputs (distinct user and card PINs); Qemu+l iterates over all the possible PIN values and counts the number of unauthorized authentications (all other input variables are fixed with arbitrary values).

The results in Table 3 (bottom group of rows) are consistent with the theoretical properties of the tools and demonstrate the interest of our method for the assessment of program vulnerabilities. PyAbd finds all the robust vulnerabilities that were already known from the Binsec/RSE runs. It better characterizes 108 programs that can be exploited by PIN combinations only, and 372 additional programs that can be exploited under favorable initial conditions. Some of these vulnerabilities have a significant number of exploitable inputs (up to 5% of PIN inputs). The analysis results of Qemu+l return a vulnerability assessment comprised between the assessments of PyAbd$^{\mathrm{O}}$ and PyAbd$^{\mathrm{P}}$, because the chosen initialization state of input values other than PINs leverages more vulnerabilities. However, Qemu+l cannot characterize the expected vulnerability on the other input variables, while this would be possible with a detailed analysis of PyAbd results.

Table 4 reports the computation times for each analysis method, including the computation times after which the first and the last constraints are returned by PyAbd. As expected, single-input simulation (Qemu) is the fastest but the most imprecise. The computation time for PyAbd (16'57") is degraded by timeouts, *i.e.* cases where no solution is found. When a solution is found, the last generated solution is returned after an average of 2'45", which is 13× as long as the average computation time of Binsec/RSE. Again, this is better than exhaustive simulation, which always has a computation time of more than one hour.

7 RELATED WORK

The link with the initial robust reachability proposal has been already lenghty discussed. We provide hereafter discussion with other abduction techniques and (standard) precondition inference.

Table 3. Analysis results for the VerifyPIN fault mutants (FISSC). Top group of rows: overall results. Bottom group: number of vulnerable mutants for which at least a given percentage of tuples (userPIN / cardPIN) lead to an incorrect authentication. Analysis methods considered: PyAbd^{O} on PIN values ; PyAbd^{P} on PIN values plus additional constraints on other input variables; Binsec; Binsec/RSE; single simulation with Qemu; and exhaustive simulation Qemu+l. Each analysis method considers a total of 4810 mutant programs.

	PyAbd^{O}	PyAbd^{P}	Binsec/RSE	Binsec	Qemu	Qemu+l
unknown	170	170	273	170	243	284
not vulnerable (0 input)	4414	4042	4419	3921	4398	4220
vulnerable ($\geq$ 1 input)	226	598	118	719	169	306
$\geq$ 0.0001%	226	598	118	–	–	306
$\geq$ 0.01%	209	582	118	–	–	281
$\geq$ 0.1%	173	514	118	–	–	210
$\geq$ 1.0%	167	472	118	–	–	199
$\geq$ 5.0%	166	471	118	–	–	196
$\geq$ 10.0%	118	401	118	–	–	148
$\geq$ 50.0%	118	401	118	–	–	135
100.0%	118	399	118	–	–	135

Table 4. Analysis times (hours:minutes:seconds) for VerifyPIN (FISSC) for the analysis methods considered in Table 3. For PyAbd$^{O/P}$, we report the complete analysis time (PyAbd$^{O/P}$), the time for returning the first constraint (PyAbd$^{O/P}_{first}$), and the time for returning the last constraint (PyAbd$^{O/P}_{last}$, *i.e.* timeouts excluded).

	PyAbd$^{O/P}$	PyAbd$^{O/P}_{first}$	PyAbd$^{O/P}_{last}$	Binsec/RSE	Binsec	Qemu	Qemu+l
average	0:16:57	0:01:53	0:02:45	0:00:13	0:00:04	0:00:01	1:08:43
median	0:01:25	0:00:46	0:00:46	0:00:06	0:00:03	0:00:01	1:11:38

General-purpose formula abduction. Extensive work has been made on efficient algorithms for general abduction in propositional logic [de Kleer 1992; Matusiewicz et al. 2011; Previti et al. 2015]. Extensions and alternative algorithms have been proposed for parts of first-order logic and complemented to handle specific theories [Bienvenu 2007; Echenim et al. 2017; Marquis 1991]. Recent approaches [Dillig and Dillig 2013; Echenim et al. 2018; Reynolds et al. 2020] aim to solve the problem while being agnostic of the underlying theory by relying on SMT solvers to verify candidates. *Our method adapts the general-purpose abduction framework for both the generation and the verification of robust precondition for satisfying reachability properties in the presence of controlled variables.* This is achieved by the three following improvements.

- First, we propose a new strategy for counter-example guidance in the presence of controlled variables, which permits to apply counter-example guidance in our context with a subset of universally quantified variables. This was not possible with the methods used in previous work;
- Second, we are the first to generate and reuse necessary constraints in this setup of constraint generation. Those constraints serve a purpose that is close to the effect obtained by using unsat-cores [Reynolds et al. 2020] but is applicable in a more general context, even with generic oracles;

- Finally, we propose a new heuristic for ordering the candidate constraints of the inference language, an idea mentioned in previous work [Echenim et al. 2018; Reynolds et al. 2020] but never explicitly studied.

Dedicated abductive techniques for program verification. Abductive reasoning has been successfully applied to program verification tasks such as including loop-invariants generation [Dillig et al. 2013; Echenim et al. 2019] and specification synthesis [Albarghouthi et al. 2016; Calcagno et al. 2009; Zhou et al. 2021]. *We are the first to apply an abductive inference technique in the context of robust reachability.*

Precondition and specification inference. There are several prior precondition inference techniques for standard reachability. White-box analyses work by reasoning on the source code structure to derive and refine satisfying solutions [Astorga et al. 2018; Cousot et al. 2013; Gulwani et al. 2008; Urban and Miné 2015]. Alternatively, black-box approaches rely on sets of examples and counter-examples to infer satisfying solutions *w.r.t.* their underlying programs [Gehr et al. 2015; Menguy et al. 2022; Padhi et al. 2016]. While they can offer satisfying performance, they can also miss solutions and are prone to overfitting.

8 CONCLUSION

We propose a novel approach based on abduction to automatically infer robust reachability constraints, i.e., sufficient conditions on a program uncontrolled input under which a target property can be reached. We specialize our approach for robust reachability of local assertions and implement it on top of an existing (robust) symbolic execution engine, demonstrating on standard benchmarks from software verification and security analysis its ability to indeed refine the notion of robust reachability in practice.

From the theoretical point of view, the approach allows to overcome a major limitation of the initial robust reachability setting *without complexifying it*, and especially without introducing any expensive quantitative reasoning. From the practical point of view, this paves the way to new verification tools able to better characterize program violations by providing the human expert with high-level feedback.

A BENCHMARKS

SV-COMP extracts. In order to analyze a wide diversity of programs, we adapt verification tasks from the SV-COMP benchmarks, written in C, to reachability problems on binary programs. For each program, a script automatically stubs the verification functions, and associates the assertion annotations in the source code to reachability targets in the compiled x86 binary. The script and the program binaries will be published for reproducibility purposes.

We triage all the programs to identify the ones of interest as follows. A first BINSEC run discovers a set of 386 programs for which the targeted location is reachable. A second run with BINSEC/RSE permits to select a subset of programs that robust symbolic execution can handle. This second set contains either 147 or 64 programs, depending on whether we use controlled variables or not. Overall, the selected programs come from the categories `bitvector`, `list`, `loops`, `nla-digbench`, `ntdrivers-simplified`, and `openssl-simplified`.

FISSC. FISSC is a collection of C programs with counter-measures against fault injection attacks. We select the VerifyPIN benchmarks, which present 10 variants of a PIN authentication procedure. The authentication proceeds by comparing a secret and an input PIN code, stored in 4 digits. The authentication is restricted to 3 attempts, but an attacker can leverage fault injection to authenticate with a wrong input PIN.

Binary programs are compiled for the ARMv7-M architecture with five compiler optimization levels (`-O0`, `-O1`, `-O2`, `-O3`, `-Os`). On each resulting program, we apply a simple fault model (single instruction skip) to each possible target fault location in the VerifyPIN function, using code mutation, resulting in 4810 distinct mutant programs. We select the 719 mutants for which Binsec can reach the target location (that is, authentication succeeds with distinct input and secret PINs). Those 719 mutants are also supported by Binsec/RSE. Simulation analysis methods (Qemu and Qemu+l) are based on QEMU version 4.2.1.

B PRACTICAL INFERENCE LANGUAGES CONSTRUCTION

For the inference languages presented in Section 6.3, the sets $\Sigma_{\mathcal{A}_8}, \Sigma_{\mathcal{A}_{32}}, \Sigma_{\mathcal{V}_8}, \Sigma_{\mathcal{V}_{32}}$ are obtained as from the symbolic execution queries. After the initial symbolic execution query ensuring that the target location is reachable (see *e.g.* Line 1 of Algorithm 2), we can extract from the witness memory state a subset of "interesting" addresses and values from which to build the sets $\Sigma_{\mathcal{A}_8}, \Sigma_{\mathcal{A}_{32}}, \Sigma_{\mathcal{V}_8}, \Sigma_{\mathcal{V}_{32}}$. We extend the sets $\Sigma_{\mathcal{A}_8}, \Sigma_{\mathcal{A}_{32}}, \Sigma_{\mathcal{V}_8}, \Sigma_{\mathcal{V}_{32}}$ by following each time a new model is generated during the execution of PyAbd. In both cases, this is done as follows: $\Sigma_{\mathcal{V}_8}$ and $\Sigma_{\mathcal{V}_{32}}$ are built by taking all the values that appear in the witness, depending on whether they are 8-bit or 32-bit values. $\Sigma_{\mathcal{A}_8}$ and $\Sigma_{\mathcal{A}_{32}}$ are built similarly by taking the memory addresses of the witness mapped to any value that is not the most frequently appearing value, since this is likely to correspond to the value assigned by the underlying solver to unconstrained variables. This distinction permits to discard a large part of the memory addresses and to keep the inference language at a reasonable size.

DATA AVAILABILITY STATEMENT

The artifact is available at https://zenodo.org/records/8421879.

ACKNOWLEDGMENTS

This work is supported by the French National Research Agency (ANR) in the framework of the Investissements d'Avenir program (ANR-10-AIRT-05, IRTNanoElec), Carnot Flexsecurity, AAPG TAVA, PEPR Cyber Secureval ANR-22-PECY-0005 and PEPR Cyber REV ANR-22-PECY-0009.

REFERENCES

Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal Specification Synthesis. In *POPL*. https://doi.org/10.1145/2837614.2837628

Angello Astorga, Siwakorn Srisakaokul, Xusheng Xiao, and Tao Xie. 2018. PreInfer: Automatic Inference of Preconditions via Symbolic Analysis. *DSN* (2018). https://doi.org/10.1109/DSN.2018.00074

Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, and Peter Stuckey. 2015. #∃SAT: Projected Model Counting. In *Theory and Applications of Satisfiability Testing – SAT*. https://doi.org/10.1007/978-3-319-24318-4_10

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *TACAS*. https://doi.org/10.1007/978-3-030-99524-9_24

Sébastien Bardin and Guillaume Girol. 2022. A Quantitative Flavour of Robust Reachability. https://doi.org/10.48550/ARXIV.2212.05244

Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2009. *Satisfiability modulo theories.* https://doi.org/10.3233/978-1-58603-929-5-825

Dirk Beyer. 2012. Competition on Software Verification (SV-COMP). In *TACAS*. https://doi.org/10.1007/978-3-642-28756-5_38

Meghyn Bienvenu. 2007. Prime Implicates and Prime Implicants in Modal Logic. In *AAAI*.

Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* (2013). https://doi.org/10.1145/2408776.2408795

Ricardo Caferra. 2013. *Logic for computer science and artificial intelligence.* John Wiley & Sons. https://doi.org/10.1002/9781118604182

Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2009. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* (2009). https://doi.org/10.1145/2049697.2049700

Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *TACAS*. https://doi.org/10.1007/978-3-540-24730-2_15

Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic Inference of Necessary Preconditions. In *VMCAI*. https://doi.org/10.1007/978-3-642-35873-9_10

Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2020. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In *IEEE SP*. https://doi.org/10.1109/SP40000.2020.00074

Adnan Darwiche. 2000. On the Tractable Counting of Theory Models and its Application to Truth Maintenance and Belief Revision. *Journal of Applied Non-Classical Logics* (2000). https://doi.org/10.3166/jancl.11.11-34

Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. 2016. Specification of concretization and symbolization policies in symbolic execution. In *ISSTA*. https://doi.org/10.1145/2931037.2931048

Johan de Kleer. 1992. An Improved Incremental Algorithm for Generating Prime Implicates. In *AAAI*. https://doi.org/10.1007/978-3-642-60211-5_9

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. https://doi.org/10.1007/978-3-540-78800-3_24

Isil Dillig and Thomas Dillig. 2013. Explain: A Tool for Performing Abductive Inference. In *CAV*. https://doi.org/10.1007/978-3-642-39799-8_46

Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive Invariant Generation via Abductive Inference. In *OOPSLA*. https://doi.org/10.1145/2509136.2509511

Adel Djoudi and Sébastien Bardin. 2015. BINSEC: Binary Code Analysis with Low-Level Regions. In *TACAS*. https://doi.org/10.1007/978-3-662-46681-0_17

Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. 2016. FISSC: A Fault Injection and Simulation Secure Collection. In *SAFECOMP*. https://doi.org/10.1007/978-3-319-45477-1_1

Mnacho Echenim, Nicolas Peltier, and Yanis Sellami. 2018. A Generic Framework for Implicate Generation Modulo Theories. In *IJCAR*. https://doi.org/10.1007/978-3-319-94205-6_19

Mnacho Echenim, Nicolas Peltier, and Yanis Sellami. 2019. Ilinva: Using Abduction to Generate Loop Invariants. In *FroCoS*. https://doi.org/10.1007/978-3-030-29007-8_5

Mnacho Echenim, Nicolas Peltier, and Sophie Tourret. 2017. Prime Implicate Generation in Equational Logic. *J. Artif. Int. Res.* 60, 1 (2017). https://doi.org/10.5555/3207692.3207711

Daniel Fremont, Markus Rabe, and Sanjit Seshia. 2017. Maximum Model Counting. *Proceedings of the AAAI Conference on Artificial Intelligence* (2017). https://doi.org/10.1609/aaai.v31i1.11138

Timon Gehr, Dimitar Dimitrov, and Martin Vechev. 2015. Learning Commutativity Specifications. In *CAV*. https://doi.org/10.1007/978-3-319-21690-4_18

Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. 2021. Not All Bugs Are Created Equal, But Robust Reachability Can Tell the Difference. In *CAV*. https://doi.org/10.1007/978-3-030-81685-8_32

Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. 2022. Introducing robust reachability. *Formal Methods in System Design* (2022). https://doi.org/10.1007/s10703-022-00402-x

Carla Gomes, Ashish Sabharwal, and Bart Selman. 2008. Model Counting. *Frontiers in Artificial Intelligence and Applications* (2008). https://doi.org/10.3233/978-1-58603-929-5-633

Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2008. Program Analysis as Constraint Solving. In *PLDI*. https://doi.org/10.1145/1379022.1375616

Max Hoffmann, Falk Schellenberg, and Christof Paar. 2021. ARMORY: Fully Automated and Exhaustive Fault Simulation on ARM-M Binaries. *IEEE Transactions on Information Forensics and Security* (2021). https://doi.org/10.1109/TIFS.2020.3027143

John R. Josephson and Susan G. Josephson. 1994. *Abductive Inference: Computation, Philosophy, Technology*. Cambridge University Press. https://doi.org/10.1017/CBO9780511530128

Seonmo Kim and Stephen McCamant. 2018. *Bit-Vector Model Counting Using Statistical Estimation*. https://doi.org/10.1007/978-3-319-89960-2_8

Patrick Koopmann, Warren Del-Pinto, Sophie Tourret, and Renate A. Schmidt. 2020. Signature-Based Abduction for Expressive Description Logics. In *KR*. https://doi.org/10.24963/kr.2020/59

Jean-Marie Lagniez and Pierre Marquis. 2019. A Recursive Algorithm for Projected Model Counting. *Proceedings of the AAAI Conference on Artificial Intelligence* (2019). https://doi.org/10.1609/aaai.v33i01.33011536

Pierre Marquis. 1991. Extending abduction from propositional to first-order logic. In *Fundamentals of Artificial Intelligence Research*. https://doi.org/10.1007/3-540-54507-7_12

Andrew Matusiewicz, Neil V. Murray, and Erik Rosenthal. 2011. Tri-Based Set Operations and Selective Computation of Prime Implicates. In *ISMIS*. https://doi.org/10.1007/978-3-642-21916-0_23

Grégoire Menguy, Sébastien Bardin, Nadjib Lazaar, and Arnaud Gotlieb. 2022. Automated Program Analysis: Revisiting Precondition Inference through Constraint Acquisition. In *IJCAI*. https://doi.org/10.24963/ijcai.2022/260

Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-Driven Precondition Inference with Learned Features. *SIGPLAN Not.* 51, 6 (jun 2016), 42–56. https://doi.org/10.1145/2980983.2908099

Alessandro Previti, Alexey Ignatiev, António Morgado, and Joao Marques-Silva. 2015. Prime Compilation of Non-Clausal Formulae. In *IJCAI*. https://doi.org/10.5555/2832415.2832524

Andrew Reynolds, Haniel Barbosa, Daniel Larraz, and Cesare Tinelli. 2020. Scalable Algorithms for Abduction via Enumerative Syntax-Guided Synthesis. In *Automated Reasoning*. https://doi.org/10.1007/978-3-030-51074-9_9

Caterina Urban and Antoine Miné. 2015. Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation. In *VMCAI*.

Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. 2018. Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation. *Journal of Hardware and Systems Security* (2018). https://doi.org/10.1007/s41635-018-0038-1

Zhe Zhou, Robert Dickerson, Benjamin Delaware, and Suresh Jagannathan. 2021. Data-Driven Abductive Inference of Library Specifications. In *OOPSLA*. https://doi.org/10.1145/3485493

Received 2023-07-11; accepted 2023-11-07