EPiC Series in Computing

Volume 100, 2024, Pages 116–126

Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning

Automatic Detection of Vulnerable Variables for CTL Properties of Programs

Naïm Moussaoui Remil[1], Caterina Urban[1], and Antoine Miné[2]

[1] Inria & ENS | PSL, Paris, France
{naim.moussaoui-remil,caterina.urban}@inria.fr
[2] Sorbonne Université, CNRS, LIP6, Paris, France
antoine.mine@lip6.fr

Abstract

We present our tool FuncTion-V for the automatic identification of the minimal sets of program variables that an attacker can control to ensure an undesirable program property. FuncTion-V supports program properties expressed in Computation Tree Logic (CTL), and builds upon an abstract interpretation-based static analysis for CTL properties that we extend with an abstraction refinement process. We showcase our tool on benchmarks collected from the literature and SV-COMP 2023.

1 Introduction

Violations of program properties pose significant risks, particularly when they can be triggered by attackers [8, 14]. This paper presents our tool FuncTion-V for the automatic identification of the *minimal* set(s) of program variables that are *vulnerable* to be controlled by an attacker to *potentially violate* a (desirable) program property, or, equivalently, the minimal variable set(s) the values of which must be controlled to *ensure* an (undesirable) program property.

Let us consider the undesirable robust reachability [8, 9] of the error location in Figure 1. Here, to make sure that the error location is reached, an attacker can either (A) control the value of x ensuring that $x > 0$ (error reachable independently of the values of y and z), or (B) control the value of y and z ensuring that $y \leq z$ (error reachable independently of the value of x). In case A, we say that x is vulnerable (while y and z are *safe*), while in case B, we say that y and z are vulnerable (while x is safe). Our tool additionally infers the sufficient conditions on the vulnerable variables that ensure the property (robust reachability of the error, in this case) independently of the values of the safe variables: $x > 0$ in case A, and $y \leq z$ in case B.

Besides robust reachability [8, 9, 18, 19], FuncTion-V more generally supports program properties expressed in Computation Tree Logic (CTL) [2] (e.g., termination [16], recurrence properties [18, 19], etc.). To improve the precision of our approach, we have equipped FuncTion-V with a conflict-driven abstraction refinement-style analysis [7].

We report on the experimentation of our tool on benchmarks from [17, 18, 4, 20, 3, 6] as well as from the termination and reachSafety categories of SV-COMP 2023.

```
1              void main (int x, int y, int z) {
2                  while( y > z ){
3                      y = y - x;
4                  }
5                  // error
6              }
7
```

Figure 1: Running Example

2 Computation Tree Logic

CTL [2] is a branching-time logic, meaning that its model of time is a tree-like structure in which each state may have several possible futures. Formulas in CTL are formed according to the following grammar:

$$\phi ::= a \mid l : a \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathrm{AX}\phi \mid \mathrm{AG}\phi \mid \mathrm{A}(\phi \,\mathrm{U}\, \phi) \mid \mathrm{EX}\phi \mid \mathrm{EG}\phi \mid \mathrm{E}(\phi \,\mathrm{U}\, \phi)$$

where a is an atomic proposition, describing properties of individual program states and $l : a$ is an atomic proposition that holds at program point l. The universal (A) and the existential (E) quantifiers allow expressing properties of all or some program executions starting in a state. The next temporal operator (X) allows expressing properties about the next state in a program execution. The globally operator (G) allows expressing properties that always (i.e., for all states) hold in a program execution. The until temporal operator (U) allows expressing a property that must hold eventually in a program execution, and another that must always hold until then. For instance, the robust reachability of the error in Figure 1 is expressed by the formula $\mathrm{A}(\mathrm{true}\, U\, \{l_5 : \mathrm{true}\})$, which means that for every execution (operator A) starting at the beginning of the program, the sequence of program states encountered during the execution must start with arbitrary states (formula true) followed ultimately (operator U) with a state at program location l_5 (formula $\{l_5 : \mathrm{true}\}$). In the following, we denote the robust reachability as $\mathrm{AF}\phi$, which is a shortcut for $\mathrm{A}(\mathrm{true}\,\mathrm{U}\,\phi)$, meaning that every program execution eventually reaches a state satisfying ϕ, i.e., $\mathrm{AF}\{l_5 : \mathrm{true}\}$ for Figure 1.

The semantics of CTL formulas is formally given by a satisfaction relation $\models$ between program states and CTL formulas. We refer to [1] for its formal definition. We write $s \models \phi$ if and only if the CTL formula ϕ holds for (the program executions starting in) the state $s \in \Sigma$.

3 Static Analysis of CTL Properties

FuncTion-V builds upon the static analysis framework proposed by Urban et al. [20]. Following the abstract interpretation theory [5], they derive a *sound and complete* semantics for proving a CTL property. Specifically, this semantics is a partial function $\Lambda_\phi \colon \Sigma \rightharpoonup \mathbb{O}$ (where $\mathbb{O}$ is the set of ordinals) defined only over the program states $s \in \Sigma$ that satisfy the CTL property ϕ, i.e., $s \models \phi$ if and only if $s \in \mathrm{dom}(\Lambda_\phi)$ [20, Theorem 1]. For instance, at the beginning of the program in Figure 1, $\Lambda_{\mathrm{AF}\{l_5:\mathrm{true}\}}$ is defined as follows:

$$\Lambda_{\mathrm{AF}\{l_5:\mathrm{true}\}}(s) = \begin{cases} v_1 & s(y) \leq s(z) \\ v_2 & s(x) > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

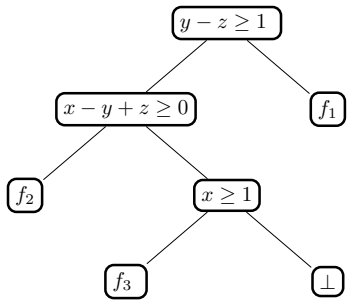

Figure 2: Decision tree inferred for $\mathrm{AF}\{l_5 : \mathrm{true}\}$ at the beginning of the program in Figure 1.

where $s(x)$ is value of the variable x in the state $s \in \Sigma$, and $v_1, v_2 \in \mathbb{O}$, meaning that $\mathrm{AF}\{l_5 : \mathrm{true}\}$ is satisfied in s if and only if $s(y) \leq s(z)$ or $s(x) > 0$. The concrete value of v_1 and v_2 is not relevant in the context of this paper so we do not discuss it (see [20] for further details).

A further *sound* abstract semantics suitable for static program analysis is derived leveraging the decision tree abstract domain $\mathcal{T}$, which is a numerical abstract domain based on piecewise-defined functions [17, 16]. Specifically, the abstraction $\Lambda^\natural_\phi \in \mathcal{T}$ of Λ_ϕ is a piecewise-defined partial function represented as a *decision tree*.

Figure 2 shows a decision tree abstracting the function $\Lambda_{\mathrm{AF}\{l_5:\mathrm{true}\}}$ shown above. The decision nodes are labeled by linear constraints (of the form $c_1 * X_1 + ... + c_k * X_k \geq c_{k+1}$) and recursively partition the space of possible values of the program variables: each constraint in a node is satisfied by the left subtree, while its negation is satisfied by the right subtree. The leaf nodes represent the value of the function corresponding to each partition. They are labeled by functions of the program variables (f_1, f_2, f_3 in Figure 2), or the special elements $\bot$ or $\top$, which explicitly represent undefined functions. The domain of the function represented by the decision tree in Figure 2 coincides with that of $\Lambda_{\mathrm{AF}\{l_5:\mathrm{true}\}}$ shown above. The partitioning is determined dynamically by the static analysis: partitions are modified by assignments and split (i.e., added) by boolean conditions and when merging control flows. A widening operator [4] extrapolates the function value in each partition and limits the size of the decision trees (and thus the number of partitions) to minimize the cost of the analysis and enforce its termination.

Let $\gamma \colon \mathcal{T} \to (\Sigma \rightharpoonup \mathbb{O})$ be the concretization function mapping decision trees to piecewise-defined partial functions. We refer to [17, 16] for its formal definition. The abstract semantics $\Lambda^\natural_\phi$ *under-approximates* the domain of Λ_ϕ (i.e., $\mathrm{dom}(\Lambda^\natural_\phi) \subseteq \mathrm{dom}(\gamma(\Lambda_\phi))$), and *over-approximates* its value (i.e., $\forall s \in \mathrm{dom}(\gamma(\Lambda^\natural_\phi)) \colon \Lambda_\phi(s) \leq \gamma(\Lambda^\natural_\phi)(s)$). In this way, $\Lambda^\natural_\phi$ yields sufficient preconditions for a state $s \in \Sigma$ to satisfy the CTL property ϕ: if $s \in \mathrm{dom}(\gamma(\Lambda^\natural_\phi))$ then $s \models \phi$ [20, Theorem 2]. In Figure 2, the value of $\Lambda_{\mathrm{AF}\{l_5:\mathrm{true}\}}$ is indeed over-approximated when $x \geq 1$, while $\mathrm{dom}(\gamma(\Lambda^\natural_\phi))$ coincides with $\mathrm{dom}(\Lambda_\phi)$ so the alarm when $y > z$ and $x \leq 0$ (where $\Lambda^\natural_\phi$ has value $\bot$ and is thus undefined) is a true positive (but that is not always the case in general).

4 Vulnerable Variable Identification

Let $\mathbb{X}$ denote the set of all program variables. We formally define when a set of variables $\mathcal{S} \subseteq \mathbb{X}$ is *safe* with respect to a CTL property ϕ. Given a variable $x \in \mathbb{X}$ and a value $v \in \mathbb{Z}$, we write $s[x \leftarrow v]$ for the program state obtained by writing the value v into the variable x in $s \in \Sigma$.

Definition 4.1 (**Safe Variables**). A set $\mathcal{S} = \{x_0 ... x_{n-1}\} \subseteq \mathbb{X}$ of program variables is *safe* with respect to a set of program states $D \subseteq \Sigma$, written $\textsc{safe}(\mathcal{S}, D)$, when there exists a program state $s \in D$ such that $\forall v_0, \ldots, v_{n-1} \in \mathbb{Z}^n$ we have $s[x_0 \leftarrow v_0, \ldots, x_{n-1} \leftarrow v_{n-1}] \in D$.

In other words, there is at least one program state that cannot escape D independently of the values of the safe variables. In the following, given a set of variables $\mathcal{S} \subseteq \mathbb{X}$ and a set of program states $D \subseteq \Sigma$, we write $D|_{\mathcal{S}}$ for the subset of all program states in D that remain in D independently of the values of the variables in $\mathcal{S}$: $D|_{\mathcal{S}} \stackrel{\text{def}}{=} \{s \in D \mid \forall v_0, \ldots, v_{n-1} \in \mathbb{Z}^n \colon s[x_0 \leftarrow v_0, \ldots, x_{n-1} \leftarrow v_{n-1}] \in D\}$. Given a CTL property ϕ, we abuse notation and write $\textsc{safe}(\mathcal{S}, \phi)$ to denote $\textsc{safe}(\mathcal{S}, \{s \in \Sigma \mid s \models \phi\})$. Note that safe variable sets are closed under inclusion, i.e., any subset of a safe variable set remains a safe variable set. We write $\overline{\textsc{safe}}(\mathcal{S}, D)$ when $\mathcal{S}$ is inclusion-wise *maximal*, i.e., $\textsc{safe}(\mathcal{S}, D) \wedge \forall \mathcal{S}' \supsetneq \mathcal{S} \colon \neg\textsc{safe}(\mathcal{S}', D)$. Inclusion-wise *maximal* sets are not unique. For example, for the program in Figure 1, we have $\overline{\textsc{safe}}(\{x\}, \mathrm{AF}\{l_5 : \mathrm{true}\})$ as well as $\overline{\textsc{safe}}(\{y, z\}, \mathrm{AF}\{l_5 : \mathrm{true}\})$.

Vulnerable variables are all variables that do not belong to a maximal set of safe variables.

Definition 4.2 (**Vulnerable Variables**). A set $\mathcal{V} \subseteq \mathbb{X}$ of program variables is *vulnerable* with respect to a set of program states $D \subseteq \Sigma$, written $\textsc{vulnerable}(\mathcal{V}, D)$, when $\overline{\textsc{safe}}(\mathbb{X} \setminus \mathcal{V}, D)$.

Namely, vulnerable variable sets are the *minimal* sets of variables the values of which must be controlled to remain in D, independently of the other (safe) variable values. Given a CTL property ϕ, we write $\textsc{vulnerable}(\mathcal{S}, \phi)$ to denote $\textsc{vulnerable}(\mathcal{S}, \{s \in \Sigma \mid s \models \phi\})$. Note that a set of variables $\mathcal{X} \subseteq \mathbb{X}$ could be *both* vulnerable and safe, if both $\overline{\textsc{safe}}(\mathcal{X}, D)$ and $\overline{\textsc{safe}}(\mathbb{X} \setminus \mathcal{X}, D)$ hold. In this case, it means that it is sufficient to control either $\mathcal{X}$ or $\mathbb{X} \setminus \mathcal{X}$ to remain in D despite the values of the other variables. This is the case for the program in Figure 1 as we have $\textsc{vulnerable}(\{y, z\}, \mathrm{AF}\{l_5 : \mathrm{true}\})$ (and $\overline{\textsc{safe}}(\{x\}, \mathrm{AF}\{l_5 : \mathrm{true}\})$) as well as $\textsc{vulnerable}(\{x\}, \mathrm{AF}\{l_5 : \mathrm{true}\})$ (and $\overline{\textsc{safe}}(\{y, z\}, \mathrm{AF}\{l_5 : \mathrm{true}\})$). Indeed, to ensure robust reachability of the error location, it is enough to control the values of y and z (such that $y \leq z$), or to control the value of x (such that $x > 0$). It is interesting to infer *all* vulnerable variables sets as this provides more information and control to the user to choose the more convenient way to achieve the same result.

Vulnerable Variable Concrete Semantics. We observe that, since the (domain of the) program semantics $\Lambda_\phi \colon \Sigma \rightharpoonup \mathbb{O}$ for a CTL property ϕ is a sufficient and necessary condition for satisfying ϕ [20, Theorem 1] (cf. Section 3), we can abstract Λ_ϕ to a *vulnerable variable semantics* $\mathcal{X}_\phi$ that identifies the vulnerable sets of program variables with respect to ϕ.

Definition 4.3 (**Vulnerable Variable Semantics**). Let $\Lambda_\phi \colon \Sigma \rightharpoonup \mathbb{O}$ be the program semantics for a CTL property ϕ. The *vulnerable variable semantics* $\mathcal{X}_\phi \in \mathcal{P}(\mathcal{P}(\mathbb{X}))$ for ϕ is defined as $\mathcal{X}_\phi \stackrel{\text{def}}{=} \alpha^{\mathcal{X}}_\phi(\Lambda_\phi)$, where $\alpha^{\mathcal{X}}_\phi(f) \stackrel{\text{def}}{=} \{\mathcal{V} \subseteq \mathbb{X} \mid \textsc{vulnerable}(\mathcal{V}, \mathrm{dom}(f))\}$.

For the program in Figure 1, $\mathcal{X}_\phi = \{\{y, z\}, \{x\}\}$.

We have that just controlling the values of any vulnerable variable set in $\mathcal{X}_\phi$ yields a sufficient condition for satisfying the CTL property ϕ.

Theorem 4.1. *Let $\mathcal{V} \in \mathcal{X}_\phi$. The program executions starting in any program state $s \in dom(\Lambda_\phi)|_{\mathbb{X} \setminus \mathcal{V}}$ satisfy the CTL property ϕ.*

For the program in Figure 1, $\{y, z\} \in \mathcal{X}_\phi$ yields the sufficient condition $\{s \in \Sigma \mid s(y) \leq s(z)\}$ (i.e., $y \leq z$) and $\{x\} \in \mathcal{X}_\phi$ yields $\{s \in \Sigma \mid s(x) > 0\}$ (i.e., $x > 0$).

Algorithm 1 Safe Variables Identification

```
1:  function DEFINITELY-SAFE(𝕏, t, S)
2:      R ← ∅
3:      for x ∈ 𝕏 do
4:          C ← S ∪ {x}
5:          if CLOSED(C) then R ← R ∪ {C}
6:          else
7:              if ¬BLOCKED(C) then
8:                  t′ ← FORGET(t, {x})
9:                  if UNDEFINED(t′) then BLOCK(C)
10:                 else
11:                     E ← UNCONSTRAINED(t′)
12:                     R′ ← DEFINITELY-SAFE(𝕏 \ E, t′, C ∪ E)
13:                     R ← R ∪ R′
14:     if R = ∅ then
15:         CLOSE(S)
16:         R = {S}
17:     return R
```

Vulnerable Variable Abstract Semantics. The vulnerable variable semantics $\mathcal{X}_\phi$ is not computable, so we leverage $\Lambda^\natural_\phi$ (cf. Section 3) to derive a sound abstraction $\mathcal{X}^\natural_\phi$.

Let $t \in \mathcal{T}$ be a decision tree representing a piecewise-defined partial function $f : \Sigma \rightharpoonup \mathbb{O}$. We observe that, if we havoc the value of a variable $x \in \mathbb{X}$ in t (e.g., we set x to a non-deterministic random value), and the resulting decision tree t' still represents a partially defined function (i.e., t' still contains leaves not labeled with $\bot$ or $\top$), then the variable x is *safe* with respect to $\mathrm{dom}(f)$ (cf. Definition 4.1). This process can be continued iteratively for each program variable, until obtaining (in general, an *under-approximation* of) a maximal safe variable set.

The function DEFINITELY-SAFE in Algorithm 1 considers all subsets of program variables to determine which ones are safe, and returns the set of all maximal safe variables sets it can find. Havoc is performed by the function FORGET invoked at Line 8 [16, Section 5.2.3]. If the resulting tree t' represents a totally undefined function (cf. Line 9), it means that the current candidate set C of variables (cf. Line 4) is not safe and so the recursive iteration should be stopped. The set C is *blocked* (cf. Line 9) to shortcut future iterations with candidate sets that are supersets of C, since they cannot be safe either (cf. Line 7). Let us consider the decision tree in Figure 2. The tree resulting after the call to FORGET$(\cdot, \{x\})$ is depicted in Figure 3. Further attempts to havoc any of the other variables result in the blocked sets $\{x, y\}$ and $\{x, z\}$. The definitely safe variable set $\{x\}$ is thus *closed* (cf. Line 15) and added to the returned result set (cf. Line 16). The decision tree resulting from either FORGET$(\cdot, \{y\})$ or FORGET$(\cdot, \{z\})$ is shown in Figure 4. Either call results in the simultaneous havoc of the other variable. Line 11 thus collects all variables that are unconstrained in t' (i.e., y and z) and adds them to the candidate definitely safe variable set in the recursive call (cf. Line 12). The call results in closing the set $\{y, z\}$ because a further havoc of x is shortcut since $\{x, y\} \subseteq \{y, z\} \cup \{x\}$ is blocked. Further iterations encountering already closed sets are also shortcut (cf. Line 5) and so the result returned by DEFINITELY-SAFE is $R = \{\{x\}, \{y, z\}\}$.

We can now define the potentially vulnerable variable semantics $\mathcal{X}^\natural_\phi$.

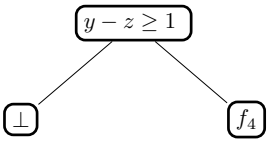

Figure 3: Figure 2 after FORGET$(\cdot, \{x\})$.

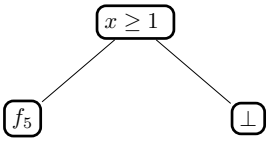

Figure 4: Figure 2 after FORGET$(\cdot, \{w\})$, $w \in \{y, z\}$.

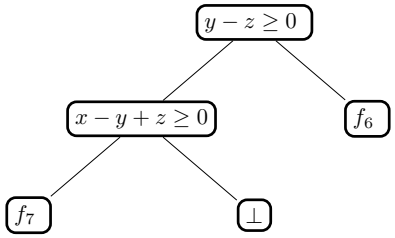

Figure 5: Decision tree inferred for the program in Figure 1 without CDA.

Definition 4.4 (**Potentially Vulnerable Variable Semantics**)**.** Let $\Lambda^\natural_\phi \in \mathcal{T}$ be the abstract program semantics for a CTL property ϕ. The *potentially vulnerable variable semantics* $\mathcal{X}^\natural_\phi \in \mathcal{P}(\mathcal{P}(\mathbb{X}))$ for ϕ is defined as $\mathcal{X}^\natural_\phi \stackrel{\text{def}}{=} \{\mathbb{X} \setminus \mathcal{S} \mid \mathcal{S} \in \text{DEFINITELY-SAFE}(\Lambda^\natural_\phi, \mathbb{X})\}$.

For any $\mathcal{W} \in \mathcal{X}^\natural_\phi$ there exists $\mathcal{V} \in \mathcal{X}_\phi$ such that $\mathcal{V} \subseteq \mathcal{W}$. Thus, we lose the minimality of vulnerable variable sets. Nonetheless, controlling the values of the potentially vulnerable variables still yields a sufficient condition for satisfying the CTL property ϕ.

Theorem 4.2. *Let $\mathcal{V} \in \mathcal{X}_\phi$. The program executions starting in any program state $s \in dom(\gamma(\text{FORGET}(\Lambda^\natural_\phi, \mathbb{X} \setminus \mathcal{V}))$ satisfy the CTL property ϕ.*

5 Conflict-Driven CTL Analysis

In order to improve the precision of the analysis, we have equipped FuncTion-V with a conflict-driven analysis (CDA) [7], generalized to handle all program properties expressed in CTL (and not just program termination as in [7]). Following an imprecise CTL analysis, the inferred decision tree defines a partition of the program states where the property is verified (tree leaves labeled with defined functions) and where the analysis is inconclusive (tree leaves labeled with $\bot$ or $\top$). The partitions corresponding to the undefined leaves of the tree form the *conflicting domain*. For instance, Figure 5 shows another possible decision tree abstraction of $\Lambda_{\text{AF}\{l_5:\text{true}\}}$ for the program in Figure 1 (less precise than that in Figure 2). The conflicting domain in this case is $y - z \geq 0 \wedge x < y - z$. Drawing inspiration from constraint programming [21], we split the conflicting domain using various cutting heuristic (e.g., splitting on the variable with the largest range of values in the conflicting domain), and repeat the CTL analysis on each resulting partition. For the tree in Figure 5, we split on x and repeat the analysis with $x \leq 0$ and $x > 0$. The process proceeds recursively until the CTL property is verified, or a threshold is exceeded. This yields the tree in Figure 2, for which FuncTion-V precisely finds the vulnerable variable sets $\{y, z\}$ and $\{x\}$. Instead, with the decision tree in Figure 5, FuncTion-V only finds $\{y, z\}$.

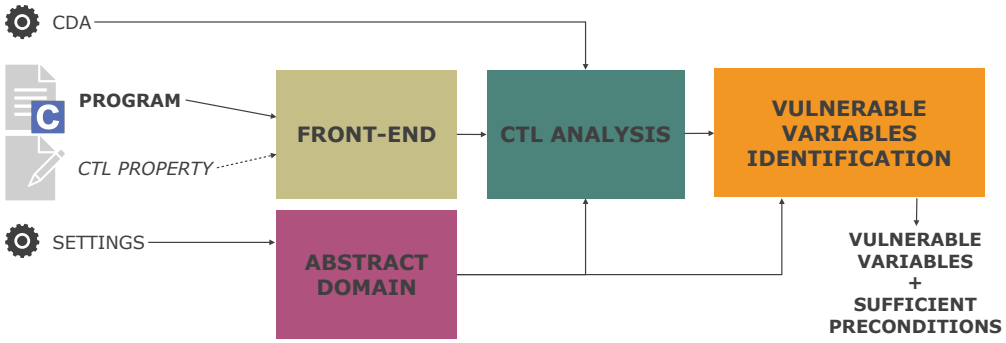

Figure 6: Overview of FuncTion-V's architecture.

6 Tool Architechture

FuncTion-V is implemented in OCaml. Figure 6 shows an overview of its architecture.

The tool takes as input a C program and a CTL property of interest (cf. Section 2). The front-end takes care of parsing the program (using an ad-hoc LR parser generated using Menhir[1]), building its internal representation, and passing it to the CTL analysis engine (cf. Section 3). The decision tree abstract domain (cf. Section 3) used by the CTL analysis can be configured (SETTINGS in Figure 6) to use different (more or less precise) domain and value representations [17] provided by the APRON numerical abstract domain library [10], as well as different (more or less sophisticated) widening heuristics [4]. The CTL analysis can optionally be refined using CDA (cf. Section 5, CDA in Figure 6).

The resulting decision tree is finally passed to the vulnerable variables identification analysis (cf. Section 4), which outputs the potentially vulnerable variables sets found and their corresponding sufficient preconditions for satisfying the given CTL property of interest.

7 Experimental Evaluation

We evaluated our tool on a number of test cases obtained from the literature [17, 18, 4, 20, 3, 6] and from the termination and reachSafety categories of the 11th International Competition on Software verification (SV-COMP 2023). The experiments were conducted on a 64-bit 8-Core CPU (AMD® Ryzen 7 pro 5850u) with 16 GB of RAM on Ubuntu 20.04 with a timeout of 120s. We configured the analysis to use decision tree with polyhedral constraints for all tests, and we evaluate the impact of the CDA extension on its precision.

For each test case, FuncTion-V found between zero and three potentially vulnerable variables sets. We present in Figure 7 the number of programs for which FuncTion-V found no potentially vulnerable set (i.e., the property of interest always holds), and for which there are n potentially vulnerable sets (i.e., one of these set must be controlled to ensure the property). We additionally detail the total (for all programs) and average (for each program) analysis time, and the average number of lines of code. When the CTL analysis returns a totally undefined function (i.e., a decision tree with only $\bot$ or $\top$ leaves), FuncTion-V returns a single potentially vulnerable variable set containing all program variables, but no sufficient condition can be found to ensure the property of interest. These cases are indicated in parentheses in Figure 7.

[1] https://gallium.inria.fr/~fpottier/menhir/

	Vulnerable Sets	Number of Programs	Total Time	Average Time	Average LOC
no-CDA	0	146	15 s	0.1 s	79
	1	165 (+15)	21 s	0.1 s	22
	2	30	10 s	0.33 s	24
	3	6	5 s	0.7 s	25
CDA	0	154	23 s	0.15 s	77
	1	160 (+14)	934 s	5 s	22
	2	79	432 s	15 s	25
	3	5	124 s	25 s	26

Figure 7: Number of potentially vulnerable variable sets.

	Termination	Robust Reachability	CTL
no-CDA	17%	26%	32%
CDA	16%	22%	28%

Figure 8: Average minimum percentage of potentially vulnerable variables.

In Figure 8, tests are categorized based on the property of interest: *termination* (242 tests), *robust reachability* (95 tests), and *CTL* (130 tests) for other CTL properties. For each category, we report the average minimum percentage of (potentially vulnerable) variables that FuncTion-V identified (excluding the cases in which no sufficient condition can be found).

Figure 7 and Figure 8 show that CDA improves the precision of FuncTion-V, allowing it to identify *fewer* and *smaller* potentially vulnerable variables sets. This precision improvement comes with a non-negligible (but still small on average) performance cost (cf. Figure 7).

The full experimental results for each test case can be found as part of our artifact[2].

8 Related Work

The program property of *robust reachability* [8, 9] refines the standard notion of reachability of a bug given a partition of the program variables in a *controlled* and an *uncontrolled* set: a bug is robustly reachable if it is reachable whatever the values of the uncontrolled variables. FuncTion-V can analyze robust reachability properties expressible as a CTL formula of the form AFϕ. The vulnerable variables sets identified by FuncTion-V identify the controlled program variables, and the corresponding sufficient preconditions inferred by FuncTion-V yield the space of values for the *controlled* program variables that ensure the robust reachability property, i.e., instead of returning a single bug witness like the symbolic execution framework proposed by [8, 9], FuncTion-V returns a set of bug witnesses. Instead, the approach recently proposed by Sellami et al. [15] infers constraints on the *uncontrolled* program variables.

The non-exploitability analysis proposed by Parolini and Miné [14] uses a combination of forward taint and value analyses by abstract interpretation to infer the absence of (potentially) exploitable runtime errors. A runtime error is (potentially) exploitable if it is triggered by a subset of the variables initialized by an external *input* (that models a potential attacker). Our tool is able to prove non-exploitability by analyzing the absence of (runtime) errors (as a CTL property). If the *input* variables are safe then the runtime errors are non-exploitable (i.e., there is no choice of values for the input variables that ensures the presence of runtime errors).

[2] https://zenodo.org/records/10964500

The under-approximating analysis proposed by Miné [13] (recently extended in [12]) aims at finding sufficient preconditions for (un)desired program properties. It is used by the CTL analysis in FuncTion-V to handle existential CTL properties. We could also use it by itself, as an alternative to the CTL analysis, before the vulnerable variable identification. In this case, however, we would lose the sensitivity to termination, i.e., the (un)desired program property is satisfied provided that the relevant program point(s) are reached by the program execution.

9 Conclusion and Future Work

In this work, we have presented a new static analyzer called FuncTion-V for the automatic identification of vulnerable program variables that could be controlled to ensure an (un)desirable CTL property. We have derived our static analysis within the framework of abstract interpretation, building upon a static analysis for CTL program properties that we extended with a conflict-driven abstraction refinement-style analysis. Using experimental evidence on a wide variety of benchmarks, we showed that FuncTion-V is able to discover vulnerable variables and the corresponding sufficient conditions to ensure the CTL property of interest. It remains for future work to extend the analysis to support a broader range of programs (e.g., array- and pointer-manipulating programs). We also plan to investigate a combination of the non-exploitability analysis of Parolini and Miné [14] with our CTL and vulnerable variable analysis. Finally, we would like to study the applicability of our approach for identifying relevant input features in the context of machine learning explainability [11].

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[2] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*, pages 196–215. Springer, 2008.

[3] Byron Cook, Eric Koskinen, and Moshe Y. Vardi. Temporal Property Verification as a Program Analysis Task. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 333–348. Springer, 2011.

[4] Nathanaël Courant and Caterina Urban. Precise Widening Operators for Proving Termination by Abstract Interpretation. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 136–152, 2017.

[5] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[6] Daniel Dietsch, Matthias Heizmann, Vincent Langenfeld, and Andreas Podelski. Fairness Modulo Theory: A New Approach to LTL Software Model Checking. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 49–66. Springer, 2015.

[7] Vijay D'Silva and Caterina Urban. Conflict-Driven Conditional Termination. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2015.

[8] Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. Not All Bugs Are Created Equal, But Robust Reachability Can Tell the Difference. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 669–693. Springer, 2021.

[9] Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. Introducing Robust Reachability. *Formal Methods in System Design*, November 2022.

[10] Bertrand Jeannet and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.

[11] João Marques-Silva, Thomas Gerspacher, Martin C. Cooper, Alexey Ignatiev, and Nina Narodytska. Explanations for monotonic classifiers. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 7469–7479. PMLR, 2021.

[12] Marco Milanese and Antoine Miné. Generation of Violation Witnesses by Under-Approximating Abstract Interpretation. In Rayna Dimitrova, Ori Lahav, and Sebastian Wolff, editors, *Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part I*, volume 14499 of *Lecture Notes in Computer Science*, pages 50–73. Springer, 2024.

[13] Antoine Miné. Backward Under-Approximations in Numeric Abstract Domains to Automatically Infer Sufficient Program Conditions. *Sci. Comput. Program.*, 93:154–182, 2014.

[14] Francesco Parolini and Antoine Miné. Sound Abstract Nonexploitability Analysis. In Rayna Dimitrova, Ori Lahav, and Sebastian Wolff, editors, *Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part II*, volume 14500 of *Lecture Notes in Computer Science*, pages 314–337. Springer, 2024.

[15] Yanis Sellami, Guillaume Girol, Frédéric Recoules, Damien Couroussé, and Sébastien Bardin. Inference of robust reachability constraints. *Proc. ACM Program. Lang.*, 8(POPL):2731–2760, 2024.

[16] Caterina Urban. *Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs*. PhD thesis, 2015.

[17] Caterina Urban and Antoine Miné. A Decision Tree Abstract Domain for Proving Conditional Termination. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, volume 8723 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2014.

[18] Caterina Urban and Antoine Miné. Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation. In Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen, editors, *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, volume 8931 of *Lecture Notes in Computer Science*, pages 190–208. Springer, 2015.

[19] Caterina Urban and Antoine Miné. Inference of ranking functions for proving temporal properties by abstract interpretation. *Comput. Lang. Syst. Struct.*, 47:77–103, 2017.

[20] Caterina Urban, Samuel Ueltschi, and Peter Müller. Abstract Interpretation of CTL Properties. In Andreas Podelski, editor, *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*,

pages 402–422. Springer, 2018.

[21] Ghiles Ziat. *A Combination of Abstract Interpretation and Constraint Programming*. PhD thesis, 2019.