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

Guillaume Girol\textsuperscript{1}  Benjamin Farinier\textsuperscript{2}  Sébastien Bardin\textsuperscript{1}

\textsuperscript{1}CEA, List, Université Paris-Saclay, France
\textsuperscript{2}TU Wien, Vienna, Austria
Formal Verification

✓

Fix

 ptr = foo();
- ptr->bar();
+ if (ptr) ptr->bar();

return;

input

Sage

Infer

Java PathFinder

Astrée

CBMC

Core concept

Reachability

Fruitful since the 70's
Formal Verification

```c
ptr = foo();
if (ptr) ptr->bar();
return;
```

**Bad input**

- `ptr->bar();`
+ `if (ptr) ptr->bar();`

**Fix**

```c
ptr = foo();
if (ptr) ptr->bar();
return;
```
Formal Verification

Fix

```c
ptr = foo();
- ptr->bar();
+ if (ptr) ptr->bar();
return;
```

bad input

Core concept

Reachability

Fruitful since the 70’s
Problem 1 with reachability in bug finding

The number of issues found can be overwhelming

Prioritisation?
What reachability tells us: one bad input
CVE-2019-20839 is triggered whenever

- the attacker passes argument `/longpath`
- the stack canary is 0x01010180
- stack starts at 0xfff00000

Real life false positives
Formally reachable, but in reality, cannot be triggered reliably
False positives in practice

- Randomisation-based protections (stack canaries, ASLR, ...)
  Bug only works for the right randomness

- Bugs involving uninitialized memory
  Bug only works for the right initial memory

- Undefined behavior
  Even exists for compiled executables!

- Stubbing I/O or opaque functions with symbolic output
  Bug only works if the hash function is attacker-chosen

- Underspecified initial state
  Under-constrained symbolic execution
Our Goals

• A formal notion refining reachability without false positives
  Focus effort on more severe bugs first

• Amenable to automated verification
  Should be provable on compiled executables
Contributions

• Defining robust reachability, a way to draw a line between “reliably reachable” and “reachable but a false positive”.
  
  Comparison to Non-Interference, HyperLTL, ...

• Expanding Symbolic Execution and Bounded Model Checking to prove robust reachability

  Standard optimisations (path pruning, concretisation) must be revisited
  Path merging increases deduction power

• A prototype based on BINSEC, experimental evaluation and benchmark

  New insight on the exploitability of 4 CVEs
  Reasonable overhead
Defining Robust Reachability

Choose a threat Model
Partition input into controlled input $a$ and uncontrolled input $x$

$$(a, x) \vdash \ell$$ means “with inputs $a$ and $x$, the program executes code at $\ell$”

Reachability of location $\ell$

$$\exists a, x. (a, x) \vdash \ell$$

Robust Reachability of $\ell$

$$\exists a. \forall x. (a, x) \vdash \ell$$
Keeping it provable

No interactive systems Would require additional quantifier alternations
No quantitative approach Would require a new kind of model counters
(We tried briefly, and it looks prohibitively expensive)
Alternative formalisms (1): Non Interference

<table>
<thead>
<tr>
<th>Behavior does not depend on $x$</th>
<th>Implies reachability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non Interference</td>
<td>for all $a$</td>
</tr>
<tr>
<td>Robust reachability</td>
<td>for a single $a$</td>
</tr>
<tr>
<td></td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>yes</td>
</tr>
</tbody>
</table>

Non-interference + Reachability $\Rightarrow$ Robust Reachability
As a hyperproperty, robust reachability is pure hyperliveness
  • not a trace property (most studied case)
  • not \((k-)\)hypersafety \(\Rightarrow\) not solvable with self-composition

Temporal logics: Expressible in CTL, HyperLTL, but no provers for generic programming languages
As a hyperproperty, robust reachability is pure hyperliveness
  • not a trace property (most studied case)
  • not \((k\text{-})\)hypersafety \((\implies\) not solvable with self-composition\)

Temporal logics: Expressible in CTL, HyperLTL, but no provers for generic programming languages

Automatically proving robust reachability on programs requires a dedicated proof method
Proving robust reachability: universally quantified SMT formulas

Input \((a, x)\)

Unrolled program

Unrolled program

Reachability

\[ \exists a, x. \varphi(a, x) \]
Proving robust reachability: universally quantified SMT formulas

Unrolled program

Path merging
Optional in SE
Required for completeness in Robust SE

Reachability

Robust reachability

BMC
Proving robust reachability: universally quantified SMT formulas

Unrolled program

Reachability

Robust reachability

Unrolled path

BMC

SE

input \((a, x)\)

\(x > 0\)
Proving robust reachability: universally quantified SMT formulas

input \((a, x)\)

\[ x > 0 \]

Unrolled program
\[ \varphi(a, x) \]

Reachability
\[ \exists a, x. \varphi(a, x) \]

Robust reachability
\[ \exists a. \forall x. \varphi(a, x) \]

BMC

Unrolled path
\[ \varphi(a, x) \]

SE

Path merging
Optional in SE
Required for completeness in Robust SE
Proving robust reachability: universally quantified SMT formulas

- **Unrolled program**
  - \( \varphi(a, x) \)

- **Unrolled path**
  - \( \varphi(a, x) \)

- **Reachability**
  - \( \exists a, x. \varphi(a, x) \)

- **Robust reachability**
  - \( \exists a. \forall x. \varphi(a, x) \)

**Input** \((a, x)\)

**BMC**

**SE**

**Path merging**

Optional in SE

Required for completeness in Robust SE

**incomplete!**
Proving robust reachability: universally quantified SMT formulas

input \((a, x)\)

Unrolled program
\[ \varphi(a, x) \]

Reachability
\[ \exists a, x. \varphi(a, x) \]

Merged paths
\[ \varphi(a, x) \]

Robust reachability
\[ \exists a. \forall x. \varphi(a, x) \]
Proving robust reachability: universally quantified SMT formulas

**Unrolled program**

\[ \varphi(a,x) \]

**Reachability**

\[ \exists a, x. \varphi(a,x) \]

**Merged paths**

\[ \varphi(a,x) \]

**Robust reachability**

\[ \exists a. \forall x. \varphi(a,x) \]

**Path merging**

Optional in SE

Required for completeness in Robust SE
Proving robust reachability: other adoptions

assume $\psi$: $\exists a. \forall x. \psi \Rightarrow \phi$ instead of $\exists a. \forall x. \psi \land \phi$

path pruning: no extra quantifier (or lose completeness)

concretization: only works on controlled values

\[
\exists a. \forall x. \phi \xrightarrow{\text{concretize}} \exists a. \forall x. x = 90 \land \phi
\]

Other more advanced enhancements to SE probably also need to be revisited
Proof of concept implementation

• A binary-level Robust SE and Robust BMC engine based on BINSEC
• Discharges quantified SMT(arrays+bitvectors) formulas to Z3
• Evaluated against 46 reachability problems including CVE replays and CTFs

<table>
<thead>
<tr>
<th></th>
<th>BMC</th>
<th>SE</th>
<th>RBMC</th>
<th>RSE</th>
<th>RSE+path merging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>22</td>
<td>30</td>
<td>32</td>
<td>37</td>
<td>44</td>
</tr>
<tr>
<td>False positive</td>
<td>14</td>
<td>16</td>
<td>32</td>
<td>37</td>
<td>44</td>
</tr>
<tr>
<td>Inconclusive</td>
<td></td>
<td></td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Resource exhaustion</td>
<td>10</td>
<td></td>
<td>13</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Robust variants of SE and BMC

No false positives, more time-outs/memory-outs, 15% median slowdown
Case studies: 4 CVEs

CVE-2019-14192 in U-boot (remote DoS: unbounded memcpy) Robustly reachable

CVE-2019-19307 in Mongoose (remote DoS: infinite loop) Robustly reachable

CVE-2019-20839 in libvncserver (local exploit: stack buffer overflow)
  Without stack canaries: Robustly reachable
  With stack canaries: Timeout

CVE-2019-19307 in Doas (local privilege escalation: use of uninitialized memory)
  Doas = OpenBSD’s equivalent of sudo
  Depends on the configuration file /etc/doas.conf
  Use robust reachability in a more creative way
Reinterpret “controlled input” differently:

the attacker controls nothing, only executes
the sysadmin controls the configuration file: controlled input
the environment sets initial memory content etc: uncontrolled inputs

The meaning of robust reachability here

Are there configuration files which make the attacker win all the time? Yes: for example typo “permit ww” instead of “permit www”
Reinterpret “controlled input” differently:

the **attacker** controls nothing, only executes
the **sysadmin** controls the configuration file: **controlled input**
the **environment** sets initial memory content etc: **uncontrolled inputs**

---

The meaning of robust reachability here

Are there configuration files which make the attacker win all the time? **Yes:** for example typo “permit ww” instead of “permit www”

---

**Versatility of Robust Reachability**

“Controlled inputs” are not limited to “controlled by the attacker”
Related work (1): Approximating security-relevance

Automatic Exploit Generation (Avgerinos et al, 2014) (Heelan, 2013)
Related work (2): Quantitative approaches

<table>
<thead>
<tr>
<th>Qualitative: less precise</th>
<th>Quantitative: slower</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Checking</td>
<td>Probabilistic Model Checking</td>
</tr>
<tr>
<td>Non-Interference</td>
<td>Quantitative Information Flow</td>
</tr>
<tr>
<td>Robust Reachability</td>
<td>Future work?</td>
</tr>
</tbody>
</table>

A small experiment suggests solver queries would be orders of magnitude slower
Related work (3)

Flakiness (O’Hearn, 2019) Effort to get rid of tests with non deterministic outcomes: particular case of non-robustness

Fairness in Model Checking (Hart et al., 1983) Same high-level idea: filter-out “uninteresting” behaviors

Higher order test generation (Godefroid, 2011) $\forall \exists$ queries to soundly approximate opaque functions (like hash functions) in Dynamic SE
Take Away

Standard reachability leads to false positives: bugs that are technically reachable, but unreproducible in practice.

Robust reachability is a stronger property expressing that the attacker can reach the target reliably.

Can be proved by variants of SE and BMC with reasonable overhead, but usual optimisations must be revisited.

Source code: https://github.com/binsec/cav2021-artifacts
Precompiled artifacts: https://zenodo.org/record/4721753