Decision Procedures for Vulnerability Analysis

Journées du GDR GPL 2021

Benjamin Farinier

Director — Marie-Laure Potet
Supervisor — Sébastien Bardin
Formal verification aims to prove or disprove the correctness of a system with respect to a certain specification or property.

Used in a growing number of contexts:
- Cryptographic protocols
- Electronic hardware
- Software source code

Core concept: $\mathcal{M} \models \mathcal{P}$
- $\mathcal{M}$: the model of the system
- $\mathcal{P}$: the property to be checked
- $\models$: the algorithmic check

Some automated software verification techniques:
- Abstract Interpretation
- Bounded Model Checking (BMC)
- Symbolic Execution (SE)
int main () {
    int x = input();
    int y = input();
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            printf("Success!\n");
    }
    printf("Failure...\n");
}

\(\sigma \triangleq \emptyset, \Gamma \triangleq \top\)

\(\sigma \triangleq \{x = x_0, y = y_0, z = 2y_0\}\)

\(\Gamma \triangleq \top \land 2y_0 = x_0\)

\(\Gamma \triangleq \top \land 2y_0 \neq x_0\)

\(\Gamma \triangleq \top \land 2y_0 = x_0 \land x_0 \leq y_0 + 10\)

\(\Gamma \triangleq \top \land 2y_0 = x_0 \land x_0 > y_0 + 10\)

\(\Gamma \triangleq \top \land 2y_0 = x_0 \land x_0 \geq y_0 + 10\)

\(\sigma: \text{symbolic state}\)

\(\Gamma: \text{path predicate}\)

\(\{x_0 = 22, y_0 = 11\}\)
Symbolic Execution suffers several limitations...

- Path explosion
- Memory model
- Constraint solving
- Interactions with the environment

...but still leads to several successful applications

SAGE, P.Godefroid et al. ⇒ x86 instruction level SE

KLEE, C.Cadar et al. ⇒ LLVM bytecode level SE

It is now a question of applying it to vulnerability analysis
# define SIZE

void get_secret (char secr[]) {
// Retrieve the secret
}

void read_input (char src[], char dst[]) {
    int i = 0;
    while (src[i]) {
        dst[i] = src[i];
        i++;
    }
}

int validate (char secr[], char inpt[]) {
    int b = 1;
    for (int i = 0; i < SIZE; i++) {
        b &= secr[i] == inpt[i];
    }
    return b;
}

int main (int argc, char *argv[]) {
    char secr[SIZE];
    char inpt[SIZE];

    if (argc != 2) return 0;
    get_secret(secr);
    read_input(argv[1], inpt);

    if (validate(secr, inpt)) {
        printf("Success!\n");
    } else {
        printf("Failure...\n");
    }
}

GDR GPL 2021 — Benjamin Farinier — 4/37
# define SIZE

void get_secret (char secr[]) {
    // Retrieve the secret
}

void read_input (char src[], char dst[]) {
    int i = 0;
    while (src[i]) {
        dst[i] = src[i];
        i++;
    }
}

int validate (char secr[], char inpt[]) {
    int b = 1;
    for (int i = 0; i < SIZE; i++) {
        b &= secr[i] == inpt[i];
    }
    return b;
}

int main (int argc, char *argv[]) {
    char secr[SIZE];
    char inpt[SIZE];
    if (argc != 2) return 0;
    get_secret(secr);
    read_input(argv[1], inpt);
    if (validate(secr, inpt)) {
        printf("Success!
");
    } else {
        printf("Failure...
");
    }
}

Goal

Find an input such that the execution reach the “Success!” branch
#define SIZE

void get_secret (char secr[]) {
    // Retrieve the secret
}

void read_input (char src[], char dst[]) {
    int i = 0;
    while (src[i]) {
        dst[i] = src[i];
        i++;
    }
}

int validate (char secr[], char inpt[]) {
    int b = 1;
    for (int i = 0; i < SIZE; i++) {
        b &= secr[i] == inpt[i];
    }
    return b;
}

∀i.∃s.∃m0.∃p0. i: input  m: memory
    p1 ≜ p0 − SIZE
    p2 ≜ p1 − SIZE
    m1 ≜ m0 [p1..p1 + SIZE − 1] ← s
    m2 ≜ m1 [p2..p2 + N − 1] ← i
    m2 [p1..p1 + SIZE − 1] = m2 [p2..p2 + SIZE − 1]
∃i.∃s.∃m_{0}.∃p_{0}.

\[
\begin{align*}
p_{1} & \triangleq p_{0} - \text{SIZE} \\
p_{2} & \triangleq p_{1} - \text{SIZE} \\
m_{1} & \triangleq m_{0} [p_{1} \ldots p_{1} + \text{SIZE} - 1] \leftarrow s \\
m_{2} & \triangleq m_{1} [p_{2} \ldots p_{2} + N - 1] \leftarrow i \\
m_{2} [p_{1} \ldots p_{1} + \text{SIZE} - 1] & = m_{2} [p_{2} \ldots p_{2} + \text{SIZE} - 1]
\end{align*}
\]

oversimplified formula!

The real formula is about 2130 reads and 456 writes
\[ \exists i \exists s \exists m_0 \exists p_0. \]
\[
\begin{align*}
p_1 & \triangleq p_0 - \text{SIZE} \\
p_2 & \triangleq p_1 - \text{SIZE} \\
m_1 & \triangleq m_0 [p_1 \ldots p_1 + \text{SIZE} - 1] \leftarrow s \\
m_2 & \triangleq m_1 [p_2 \ldots p_2 + N - 1] \leftarrow i \\
m_2 [p_1 \ldots p_1 + \text{SIZE} - 1] &= m_2 [p_2 \ldots p_2 + \text{SIZE} - 1]
\end{align*}
\]

oversimplified formula!

The real formula is about 2130 reads and 456 writes

Unrolling-based verification techniques (BMC, SE)

- may produce huge formulas
- with a high number of reads and writes

In some extreme cases, solvers may spend \textbf{hours} on these formulas

ASPack case study: 293000 reads, 58000 writes
\[ \Rightarrow 24 \text{ hours of resolution!} \]
Sending the formula to a solver:

\[ \{ s[0..\text{SIZE}-1] = 0, i[0..\text{SIZE}-1] = 0, \ldots \} \]

“If the secret is 0, then you can choose 0 as an input.”

Sure, that is true... but a false positive in practice

- the secret will not likely be 0

\[ \Rightarrow \] the execution will not reach the “Success” branch
Sending the formula to a solver:

$$\Rightarrow \{ s[0..\text{SIZE}-1] = 0, i[0..\text{SIZE}-1] = 0, \ldots \}$$

"If the secret is 0, then you can choose 0 as an input."

Sure, that is true... but a false positive in practice

- the secret will not likely be 0
  $$\Rightarrow$$ the execution will not reach the "Success" branch

Threat models make security $\neq$ safety

A better formalization:

- We do not have control over $s$, $m_0$ and $p_0$
- These variables should be universally quantified
  $$\Rightarrow$$ This is where the problems begin...
• Symbolic Execution (SE)
  ○ under-approximation verification technique
  ○ heavily relies on SMT solvers

• Application to vulnerability analysis
  ○ requires to move from source analysis to binary analysis
  ○ modeling threat models introduces universal quantifiers

• Problems
  ○ finding a model for a $\forall$-formula is difficult
  ○ going low-level significantly increases formula size

$\Rightarrow$ The Death of SMT Solvers
Introduction

1. Model Generation for Quantified Formulas: A Taint-Based Approach

2. Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing

3. Get Rid of False Positives with Robust Symbolic Execution

4. Conclusion
Section 1

Model Generation for Quantified Formulas: A Taint-Based Approach
Model Generation for Quantified Formulas

Overview

• Challenge
  ◦ Deal with quantified-formulas and model generation
  ◦ Notoriously hard! (undecidable)

• Existing approaches
  ◦ Complete but costly for very specific theories
  ◦ Incomplete but efficient for UNSAT/UNKNOWN
  ◦ Costly or too restricted for model generation

• Our proposal
  ◦ SAT/UNKNOWN and model generation
  ◦ Incomplete but efficient, generic, theory independent
  ◦ Reuse state-of-the-art solvers as much as possible

Published in Computer Aided Verification 30th, Oxford, UK, 2018 [CAV18]
Presented in Approches Formelles dans l’Assistance au Développement de Logiciels, Grenoble, France, 2018 [AFADL18]
Model Generation for Quantified Formulas
Toy Example

```c
int main () {
    int a = input ();
    int b = input ();
    int x = rand ();
    if (a * x + b > 0) {
        analyze_me ();
    }
    else {
        ...
    }
}
```

We propose a way to infer such conditions

- Quantified reachability condition: $\forall x. ax + b > 0$
- Generalizable solutions of $ax + b > 0$ have to be independent from $x$
  - A bad solution: $a = 1 \land x = 1 \land b = 0$
  - A good solution: $a = 0 \land x = 1 \land b = 1$
- The constraint $a = 0$ is the independence condition
- Quantifier-free reachability condition: $(ax + b > 0) \land (a = 0)$
Model Generation for Quantified Formulas

Our Proposal in a Nutshell

\[ \forall x. \Phi(x, a) \rightarrow \Psi(a) \rightarrow \Phi(x, a) \land \Psi(a) \]

\texttt{sat}(x, a) \rightarrow \texttt{SAT}(x, a) \rightarrow \texttt{SAT}(a)

\texttt{unsat} \rightarrow \texttt{UNSAT}

\texttt{unknown}

\texttt{SIC inference}

\texttt{Sufficient Independent Condition}

\texttt{QF-solver}

GDR GPL 2021 — Benjamin Farinier — 12/37
Sufficient Independence Condition (SIC)

A SIC for a formula $\Phi(x, a)$ with regard to $x$ is a formula $\Psi(a)$ such that $\Psi(a) \models (\forall x. \forall y. \Phi(x, a) \iff \Phi(y, a))$.

- If $\Phi \triangleq ax + b > 0$ then $a = 0$ is a SIC$_{\Phi, x}$.
- If $\Delta \triangleq (t[a] \leftarrow b)[c]$ then $a = c$ is a SIC$_{\Delta, t}$.
- $\bot$ is always a SIC, but a useless one...

Model generalization

- Let $\Phi(x, a)$ a formula and $\Psi(a)$ a SIC$_{\Phi, x}$.
- If there exists an interpretation $\{x, a\}$ such that $\{x, a\} \models \Psi(a) \land \Phi(x, a)$, then $\{a\} \models \forall x. \Phi(x, a)$. 
Weakest Independence Condition (WIC)

A WIC for a formula $\Phi(x, a)$ with regard to $x$ is a $\text{SIC}_{\Phi,x} \, \Pi$ such that, for any other $\text{SIC}_{\Phi,x} \, \Psi$, $\Psi \models \Pi$.

- Both $\text{SIC} \, a = 0$ and $a = c$ presented earlier are WIC.
- $\Omega \triangleq \forall x. \forall y. (\Phi(x, a) \iff \Phi(y, a))$ is always a $\text{WIC}_{\Phi,x}$, but involves quantifiers.
- A formula $\Pi$ is a $\text{WIC}_{\Phi,x}$ if and only if $\Pi \equiv \Omega$.

Model specialization

- Let $\Phi(x, a)$ a formula and $\Pi(a)$ a $\text{WIC}_{\Phi,x}$.
- If there exists an interp. $\{a\}$ such that $\{a\} \models \forall x. \Phi(x, a)$, then $\{x, a\} \models \Pi(a) \land \Phi(x, a)$ for any valuation $x$ of $x$. 
Function inferSIC(Φ, x):

Input: Φ a formula and x a set of targeted variables
Output: Ψ a SICΦ,x

either Φ is a constant

return ⊤

either Φ is a variable v

return v ∉ x

either Φ is a function f (φ₁,…,φₙ)

Let ψᵢ ≜ inferSIC(φᵢ, x) for all i ∈ {1,…,n}
Let Ψ ≜ theorySIC(f,(φ₁,…,φₙ),(ψ₁,…,ψₙ),x)

return Ψ ∨ \bigwedgeᵢ ψᵢ

Syntactic part:
a and b indepₓ ↼ f(a, b) indepₓ

Semantic part:
a indepₓ and a = 0 ↼ a · * indepₓ
Model Generation for Quantified Formulas

Taint-based SIC inference

**Proposition**

- If \( \text{theorySIC}(f, \phi_i, \psi_i, x) \) computes a \( \text{SIC}_f(\phi_i), x \), then \( \text{inferSIC}(\Phi, x) \) computes a \( \text{SIC}_{\Phi}, x \).

**Function inferSIC(\Phi, x):**

- **Input:** \( \Phi \) a formula and \( x \) a set of targeted variables
- **Output:** \( \Psi \) a \( \text{SIC}_{\Phi}, x \)

  - either \( \Phi \) is a constant
    - return \( \top \)
  - either \( \Phi \) is a variable \( v \)
    - return \( v \notin x \)
  - either \( \Phi \) is a function \( f(\phi_1, \ldots, \phi_n) \)
    - Let \( \psi_i \triangleq \text{inferSIC}(\phi_i, x) \) for all \( i \in \{1, \ldots, n\} \)
    - Let \( \Psi \triangleq \text{theorySIC}(f, (\phi_1, \ldots, \phi_n), (\psi_1, \ldots, \psi_n), x) \)
    - return \( \Psi \lor \bigwedge_i \psi_i \)

  syntactic part
  - \( a \) and \( b \) indep_x \( \leadsto f(a, b) \) indep_x

  semantic part
  - \( a \) indep_x and \( a = 0 \) \( \leadsto a \cdot \ast \) indep_x
theorySIC defined as a recursive function

\[(a \Rightarrow b)^* \triangleq (a^* \wedge a = \bot) \lor (b^* \wedge b = \top)\]
\[(a \wedge b)^* \triangleq (a^* \wedge a = \bot) \lor (b^* \wedge b = \bot)\]
\[(a \vee b)^* \triangleq (a^* \wedge a = \top) \lor (b^* \wedge b = \top)\]
\[(\text{ite } c \ a \ b)^* \triangleq (c^* \wedge \text{ite } c \ a^* \ b^*) \lor (a^* \wedge b^* \wedge a = b)\]
\[(a_n \wedge b_n)^* \triangleq (a_n^* \wedge a_n = 0_n) \lor (b_n^* \wedge b_n = 0_n)\]
\[(a_n \vee b_n)^* \triangleq (a_n^* \wedge a_n = 1_n) \lor (b_n^* \wedge b_n = 1_n)\]
\[(a_n \times b_n)^* \triangleq (a_n^* \wedge a_n = 0_n) \lor (b_n^* \wedge b_n = 0_n)\]
\[(a_n \ll b_n)^* \triangleq (b_n^* \wedge b_n \geq n)\]
\[((a[i] \leftarrow e)[j])^* \triangleq (\text{ite } (i = j) \ e \ (a[j]))^*\]
\[\triangleq ((i = j)^* \wedge (\text{ite } (i = j) \ e^* \ (a[j])^*)) \lor (e^* \wedge (a[j])^* \wedge (e = a[j]))\]
\[\triangleq (i^* \wedge j^* \wedge (\text{ite } (i = j) \ e^* \ (a[j])^*)) \lor (e^* \wedge (a[j])^* \wedge (e = a[j]))\]
## Experimental Evaluation

### Best approaches

<table>
<thead>
<tr>
<th></th>
<th>Z3</th>
<th>Btor•</th>
<th>Btor• &gt; Z3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SMT-LIB</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAT</td>
<td>261</td>
<td>399</td>
<td>485</td>
</tr>
<tr>
<td># UNSAT</td>
<td>165</td>
<td>N/A</td>
<td>165</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>843</td>
<td>870</td>
<td>619</td>
</tr>
<tr>
<td>total time</td>
<td>270150</td>
<td>350</td>
<td>94610</td>
</tr>
<tr>
<td><strong>BINSEC</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAT</td>
<td>953</td>
<td>1042</td>
<td>1067</td>
</tr>
<tr>
<td># UNSAT</td>
<td>319</td>
<td>N/A</td>
<td>319</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>149</td>
<td>379</td>
<td>35</td>
</tr>
<tr>
<td>total time</td>
<td>64761</td>
<td>1152</td>
<td>1169</td>
</tr>
</tbody>
</table>

### GRUB example

<table>
<thead>
<tr>
<th></th>
<th>Z3</th>
<th>Btor•</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT</td>
<td>1</td>
<td>540</td>
</tr>
<tr>
<td># UNSAT</td>
<td>42</td>
<td>N/A</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>852</td>
<td>355</td>
</tr>
<tr>
<td>total time</td>
<td>159765</td>
<td>16732</td>
</tr>
</tbody>
</table>

### Complementarity with existing solvers (SAT instances)

<table>
<thead>
<tr>
<th></th>
<th>CVC4•</th>
<th>Z3•</th>
<th>Btor•</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SMT-LIB</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVC4</td>
<td>-10 +168 [252]</td>
<td>-10 +325 [409]</td>
<td></td>
</tr>
<tr>
<td>Z3</td>
<td>-119 +224 [485]</td>
<td>-86 +224 [485]</td>
<td></td>
</tr>
<tr>
<td><strong>BINSEC</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVC4</td>
<td>-25 +28 [979]</td>
<td>-25 +116 [1067]</td>
<td></td>
</tr>
<tr>
<td>Z3</td>
<td>-25 +114 [1067]</td>
<td>-25 +114 [1067]</td>
<td></td>
</tr>
</tbody>
</table>

solver•: solver enhanced with our method

---

Boolector: an efficient QF-solver for bitvectors and arrays
Section 2

Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing
Arrays Made Simpler
Overview

- **Challenge**
  - Array theory useful for modelling memory or data structures...
  - ...but a bottleneck for resolution of large formulas (BMC, SE)

- **Existing approaches**
  - General decision procedures for the theory of arrays
  - Dedicated handling of arrays inside tools

- **Our proposal**
  - FAS, an efficient simplification for array theory
  - Improves existing solvers

Published in Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 2018 [LPAR18]
Presented in Journées Francophones des Langages Applicatifs, Banyuls-sur-Mer, France, 2018 [JFLA18]
Two basic operations on arrays

- **Reading** in $a$ at index $i \in \mathcal{I}$: $a[i]$
- **Writing** in $a$ an element $e \in \mathcal{E}$ at index $i \in \mathcal{I}$: $a[i] \leftarrow e$

\[
\cdot [\cdot] : \text{Array } \mathcal{I} \times \mathcal{E} \rightarrow \mathcal{I} \rightarrow \mathcal{E} \\
\cdot [\cdot] \leftarrow \cdot : \text{Array } \mathcal{I} \times \mathcal{E} \rightarrow \mathcal{I} \rightarrow \mathcal{E} \rightarrow \text{Array } \mathcal{I} \times \mathcal{E}
\]

ROW-axiom: $\forall a \ i \ j \ e. (a[i] \leftarrow e)[j] = \begin{cases} e & \text{if } i = j \\ a[j] & \text{otherwise} \end{cases}$

---

**Prevalent in software analysis**
- Modelling memory
- Abstracting data structure (map, queue, stack...)

**Hard to solve**
- NP-complete
- Read-Over-Write (ROW) may require case-splits
Unrolling-based verification techniques (BMC, SE)
- may produce huge formula
- high number of reads and writes

In some extremes cases, solvers may spend hours on these formulas

Without proper simplification, array theory might become a bottleneck for resolution

What should we simplify? **Read-Over-Write (ROW)!**
Arrays Made Simpler
ROW Simplification

An example coming from binary analysis

\[ \begin{align*}
\text{esp}_0 & : \text{BitVec16} \\
\text{mem}_0 & : \text{Array BitVec16 BitVec16} \\
\text{assert} (\text{esp}_0 > 61440) \\
\text{mem}_1 & \triangleq \text{mem}_0 [\text{esp}_0 - 16] \leftarrow 1415 \\
\text{esp}_1 & \triangleq \text{esp}_0 - 64 \\
\text{eax}_0 & \triangleq \text{mem}_1 [\text{esp}_1 + 48] \\
\text{assert} (\text{mem}_1 [\text{eax}_0] = 9265)
\end{align*} \]

These simplifications depend on two factors

- The equality check procedure
  - verify that \( \text{esp}_1 + 48 = \text{esp}_0 - 16 \)
  - \( \Rightarrow \text{precise reasoning: base normalization + abstract domains} \)

- The underlying representation of an array
  - remember that \( \text{mem}_1 [\text{esp}_1 + 48] = 1415 \)
  - \( \Rightarrow \text{scalability issue: list-map representation} \)
Arrays Made Simpler
Improving scalability: list-map representation

How to update
Given a write of e at index i
- Is i comparable with indices of elements in the head?
- If so add (i, e) in this map
- Else add a new head map containing only (i, e)

How to simplify ROW
Given a read at index j
- Is j comparable with indices of elements in the head?
- If so, look for (i, e) with i=j
  - if succeeds then return e
  - else recurse on next map
- Else stop

GDR GPL 2021 — Benjamin Farinier — 23/37
Arrays Made Simpler
Precise reasoning: base normalization and abstract domains

Propagate “variable+constant” terms
• If \( y \triangleq z + 1 \) then \( x \triangleq y + 2 \) \( \rightsquigarrow \) \( x \triangleq z + 3 \)
• Together with associativity, commutativity...
⇒ Reduce the number of bases

Associate to every indices \( i \) an abstract domain \( i^\# \)
• If \( i^\# \sqcap j^\# = \bot \) then \((a[i] \leftarrow e)[j] = a[j] \)
• Integrated in the list-map representation
⇒ Prove disequality between different bases
- 6,590 x 3 medium-size formulas from static SE
- \textbf{TIMEOUT} = 1,000 seconds

<table>
<thead>
<tr>
<th>simpl. time</th>
<th>#TIMEOUT and resolution time</th>
<th>#ROW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Booletor</td>
<td>Yices</td>
</tr>
<tr>
<td>default</td>
<td></td>
<td></td>
</tr>
<tr>
<td>concrete</td>
<td>61</td>
<td>0</td>
</tr>
<tr>
<td>FAS</td>
<td>85</td>
<td>0</td>
</tr>
<tr>
<td>FAS-итv</td>
<td>111</td>
<td>0</td>
</tr>
<tr>
<td>interval</td>
<td></td>
<td></td>
</tr>
<tr>
<td>default</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>FAS</td>
<td>99</td>
<td>0</td>
</tr>
<tr>
<td>FAS-итv</td>
<td>118</td>
<td>0</td>
</tr>
<tr>
<td>symbolic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>default</td>
<td>61</td>
<td>0</td>
</tr>
<tr>
<td>FAS</td>
<td>91</td>
<td>0</td>
</tr>
<tr>
<td>FAS-итv</td>
<td>111</td>
<td>0</td>
</tr>
</tbody>
</table>
• 29 x 3 very large formulas from dynamic SE
• TIMEOUT = 1,000 seconds

<table>
<thead>
<tr>
<th>simpl. time</th>
<th>#TIMEOUT and resolution time</th>
<th>#ROW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Booletor</td>
<td>Yices</td>
</tr>
<tr>
<td></td>
<td>44</td>
<td>10</td>
</tr>
<tr>
<td>concrete</td>
<td>1,108</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>196</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>210</td>
<td>4</td>
</tr>
<tr>
<td>interval</td>
<td>44</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>222</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>231</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>237</td>
<td>12</td>
</tr>
<tr>
<td>symbolic</td>
<td>40</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>187</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>194</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>11</td>
</tr>
</tbody>
</table>
Huge formula obtained from the ASPack packing tool
- 293,000 rows
- 24 hours of resolution!

Using FAS
- \#ROW reduced to 2,467
- 14 sec for resolution
- 61 sec for preprocessing

Using list representation
- Same result with a bound of 385,024 and beyond...
- ...but 53 min preprocessing
Section 3

Get Rid of False Positives with Robust Symbolic Execution
• Symbolic Execution (SE)
  ○ under-approximation verification technique
  ○ heavily relies on SMT solvers
  ○ should be exempt of false positives

• In practice, false positives exist
  ○ misspecified abstractions, initial state...
  ○ some ad hoc workarounds, no real solution

• Our proposal: Robust Symbolic Execution
  ○ distinguish between controlled and uncontrolled inputs
  ○ robust solutions are independent of uncontrolled inputs
  ○ practical application of [CAV18] and [LPAR18]

Presented in Journées Francophones des Langages Applicatifs, Les Rousses, France, 2019 [JFLA19]
#define SIZE

void get_secret (char secr[]) {
    // Retrieve the secret
}

void read_input (char src[], char dst[]) {
    int i = 0;
    while (src[i]) {
        dst[i] = src[i];
        i++;
    }
}

int validate (char secr[], char inpt[]) {
    int b = 1;
    for (int i = 0; i < SIZE; i++) {
        b &= secr[i] == inpt[i];
    }
    return b;
}

∀i.∃s.∃m_0.∃p_0.
    p_1 \triangleq p_0 - SIZE
    p_2 \triangleq p_1 - SIZE
    m_1 \triangleq m_0 [p_1..p_1 + SIZE - 1] \leftarrow s
    m_2 \triangleq m_1 [p_2..p_2 + N - 1] \leftarrow i
    m_2 [p_1..p_1 + SIZE - 1] = m_2 [p_2..p_2 + SIZE - 1]
\[\exists i. \exists s. \exists m_0. \exists p_0.\]
\[
p_1 \triangleq p_0 - \text{SIZE} \\
p_2 \triangleq p_1 - \text{SIZE} \\
m_1 \triangleq m_0 [p_1..p_1 + \text{SIZE} - 1] \leftarrow s \\
m_2 \triangleq m_1 [p_2..p_2 + N - 1] \leftarrow i \\
m_2 [p_1..p_1 + \text{SIZE} - 1] = m_2 [p_2..p_2 + \text{SIZE} - 1]
\]

Sending the formula to a solver:
\[
\Rightarrow \{ s[0..\text{SIZE} - 1] = 0, i[0..\text{SIZE} - 1] = 0, \ldots \}
\]
- This is a false positive

A better formalization: Robust SE
- We do not have control over \(s, m_0\) and \(p_0\)
- These variables should be universally quantified
\[ \exists i. \forall s. \forall m_0. \forall p_0. \]
\[ p_1 \triangleq p_0 - \text{SIZE} \]
\[ p_2 \triangleq p_1 - \text{SIZE} \]
\[ m_1 \triangleq m_0[p_1..p_1 + \text{SIZE} - 1] \leftarrow s \]
\[ m_2 \triangleq m_1[p_2..p_2 + N - 1] \leftarrow i \]
\[ m_2[p_1..p_1 + \text{SIZE} - 1] = m_2[p_2..p_2 + \text{SIZE} - 1] \]

Problems:

- finding a model for a \( \forall \)-formula is difficult
- going low-level significantly increases formula size

\( \Rightarrow \) The Death of SMT Solvers
∃i. ∃s. ∃m_0. ∀p_0.
\[p_1 \triangleq p_0 - \text{SIZE}\]
\[p_2 \triangleq p_1 - \text{SIZE}\]
\[m_1 \triangleq m_0 \[p_0 - \text{SIZE}..p_0 - 1\] \leftarrow s\]
\[m_2 \triangleq m_1 \[p_0 - 2 \cdot \text{SIZE}..p_0 - 2 \cdot \text{SIZE} + N - 1\] \leftarrow i\]

\[i[0..\text{SIZE} - 1] = i[\text{SIZE}..2 \cdot \text{SIZE} - 1]\]
\[\land N \geq 2 \cdot \text{SIZE}\]

Problems:

- finding a model for a ∀-formula is difficult [CAV18]
- going low-level significantly increases formula size [LPAR18]

⇒ The Death of SMT Solvers

For example with \(\text{SIZE} = 8\),

- input abcdefghabcdefgh leads to the “Success!” branch
- buffer overflow in read_input
- set of crackme challenges
- compare true and false positives

<table>
<thead>
<tr>
<th></th>
<th>SE classic</th>
<th>SE robust</th>
<th>SE robust + elim.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>true positives</td>
<td>false positives</td>
<td>unknown</td>
</tr>
<tr>
<td>Boolector</td>
<td>12</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>CVC4</td>
<td>7</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Yices</td>
<td>7</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>Z3</td>
<td>12</td>
<td>12</td>
<td>0</td>
</tr>
</tbody>
</table>
Back to 28: GRUB2 Authentication Bypass

- Original version: press Backspace 28 times to get a rescue shell
- Case study: same vulnerable code turned into a crackme challenge

- SE classic: incorrect solution
- SE robust: solvers TIMEOUT
- SE robust + elim.: correct solution in 80s
- SE robust + elim. + simpl.: correct solution in 30s
Section 4

Conclusion
• Symbolic Execution (SE)
  ○ under-approximation verification technique
  ○ heavily relies on SMT solvers

• Application to vulnerability analysis
  ○ requires to move from source analysis to binary analysis
  ○ modeling threat models introduces universal quantifiers

• Problems
  ○ finding a model for a $\forall$-formula is difficult
  ○ going low-level significantly increases formula size
  $\Rightarrow$ The Death of SMT Solvers
Conclusion
Contributions

1 Model Generation for Quantified Formulas
   ◦ Proposed a novel and generic taint-based approach
   ◦ Proved its correctness and its efficiency
   ◦ Presented an implementation for arrays and bit-vectors
   ◦ Evaluated on SMT-LIB and formulas generated by Symbolic Execution

2 Arrays Made Simpler
   ◦ Presented FAS, a simplification dedicated to the theory of arrays
   ◦ Geared at eliminating ROW, based on a dedicated data structure, original simplifications and low-cost reasoning
   ◦ Evaluated in different settings on very large formulas

3 Robust Symbolic Execution
   ◦ Highlighted the problem of false positives in classic Symbolic Execution
   ◦ Introduced formally the framework of Robust Symbolic Execution
   ◦ Implemented a proof of concept in the binary analyser BINSEC

Not all bugs are created equal, but robust reachability can tell the difference.
In *CAV 2021, Virtual, July 18-24, 2021*.

- Formal definition of Robust Reachability, application to SE and BMC
- Adaptation of standard optimizations to Robust Reachability
- Evaluation against 46 reachability problems including CVE replays and CTFs

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

- Universal quantification of formulas has a cost, but not so high.
- RSE+ (robust SE with path merging) is 15% slower than SE in median, but with large outliers.