Decision Procedures for Vulnerability Analysis

June 24, 2020

Benjamin Farinier

Director	—	Marie-Laure Potet
Supervisor	—	Sébastien Bardin
Reviewer	—	Thomas Jensen
Reviewer	—	Sylvain Conchon
Examiner	—	Mihaela Sighireanu
Examiner	—	Jean Goubault-Larrecq
Examiner	—	Roland Groz

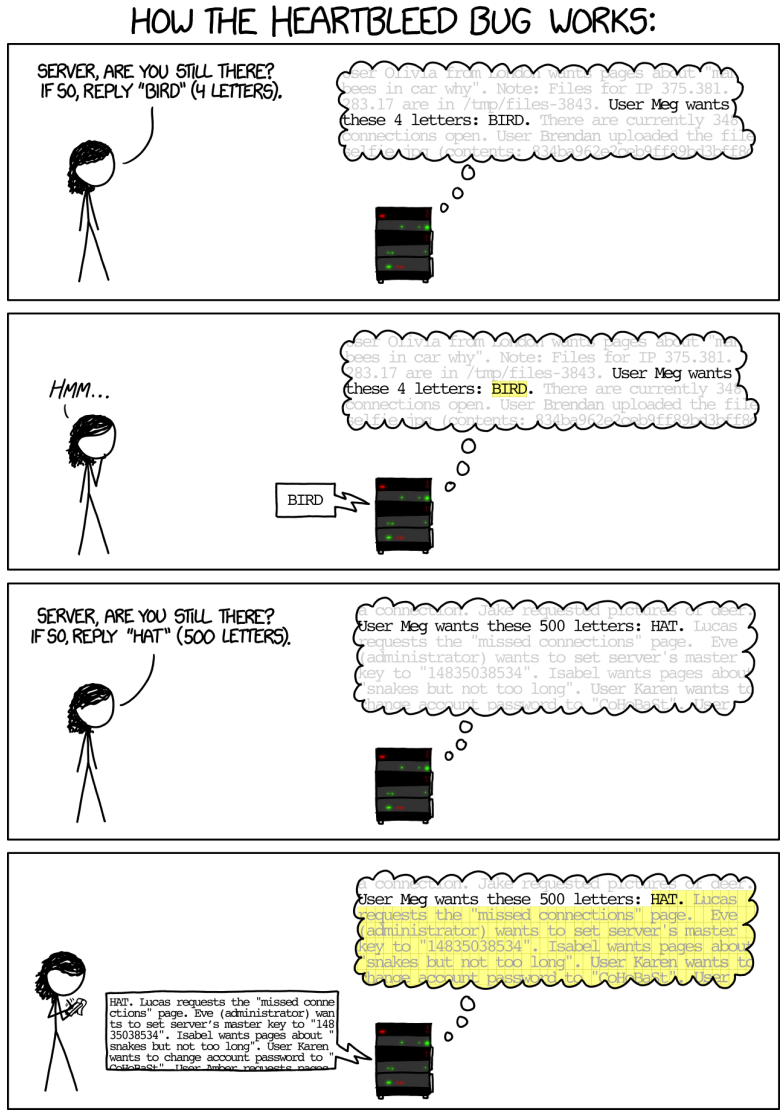
On April 7, 2014, the Heartbleed software vulnerability is made public

- between 24% and 55% of "secure" servers were affected
- due to a bug introduced on March 14, 2012 in the OpenSSL cryptography library
- allows an attacker to read secret data form the memory of a vulnerable server

The vulnerability is caused by a *buffer overflow*

- a kind of vulnerability known since 1972
- already exploited by the Morris computer worm in 1988

Long-standing causes known as dangerous are still the source of major vulnerabilities
$\Longrightarrow$ need for research in formal verification



Formal Verification

Formal verification aims to prove or disprove the correctness of a system with respect to a certain specification or property

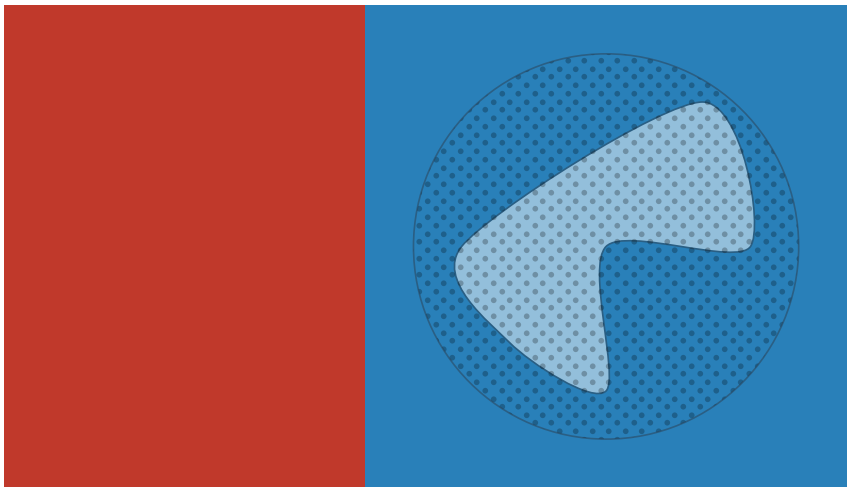
Used in a growing number of contexts

- Cryptographic protocols
- Electronic hardware
- Software source code

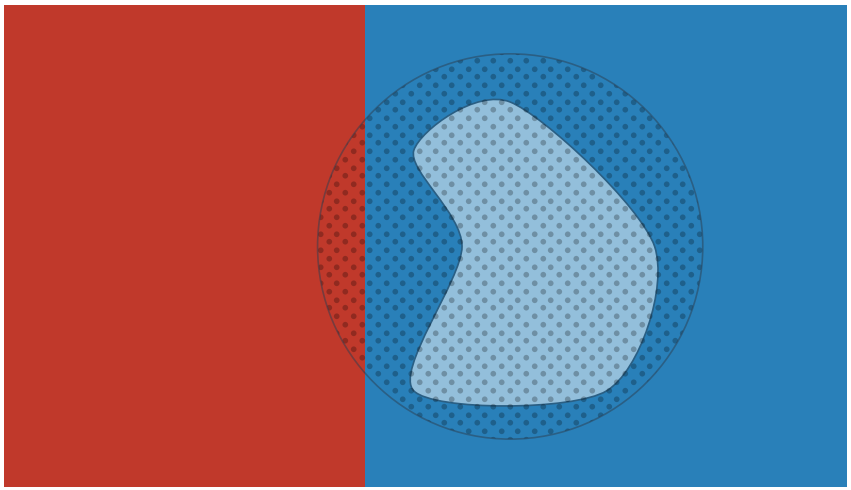
Core concept: $\mathcal{M} \models \mathcal{P}$

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

Over-Approximation



correct behaviours
incorrect behaviours



object behaviours
model behaviours

With an over-approximating model:

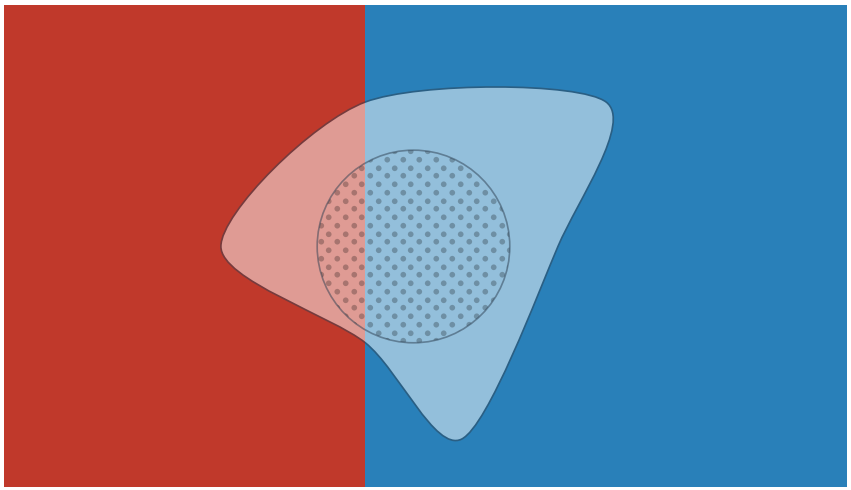
- All object behaviors are captured by the model
- But some model behaviors are not realizable by the object

$\Rightarrow$ False positive

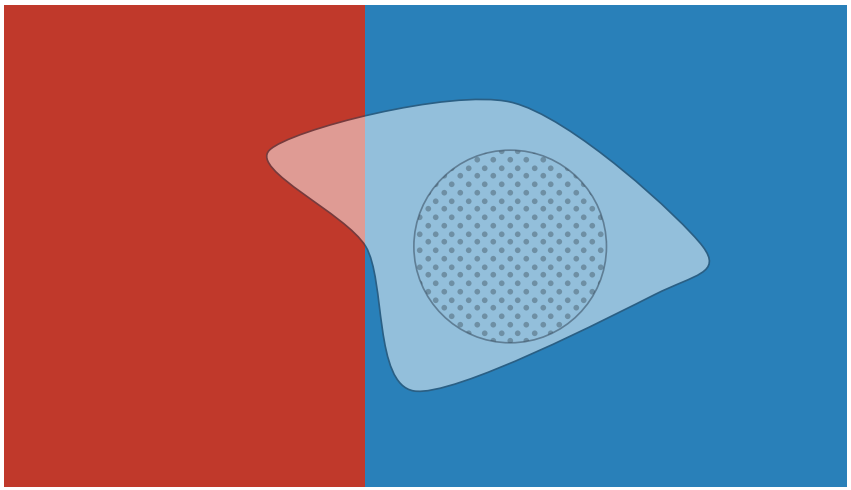
Some over-approximation techniques:

- Abstract Interpretation
- Hoare Logic
- ...

Under-Approximation



correct behaviours
incorrect behaviours



object behaviours
model behaviours

With an under-approximating model:

- All model behaviors are realizable by the object
- But some object behaviors are not captured by the model

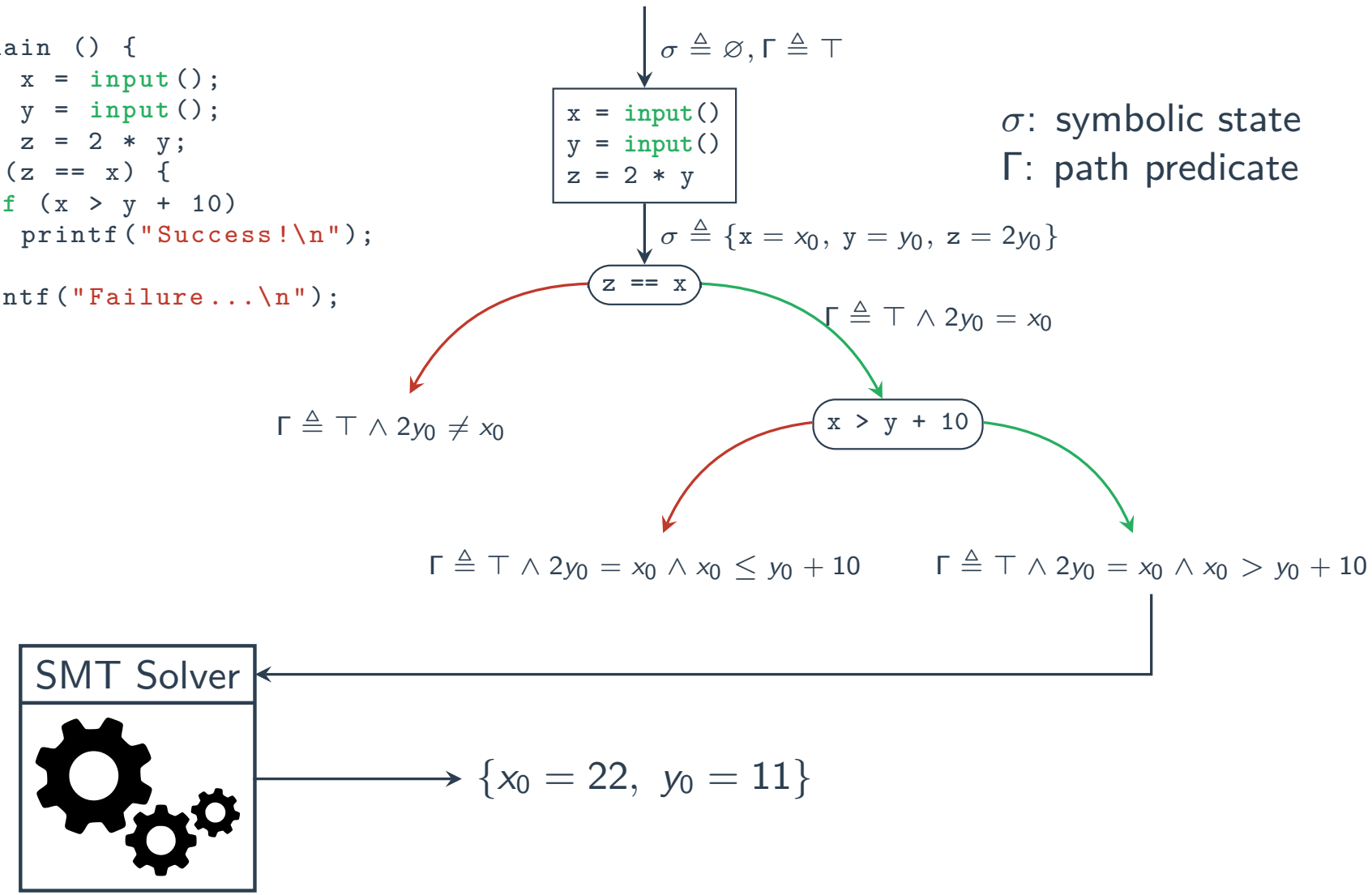
$\Rightarrow$ False negative

Some under-approximation techniques:

- Bounded Model Checking (BMC)
- Symbolic Execution (SE)
- ...

Symbolic Execution

```
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");
}
```



Limitations and Successes

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.

$\Rightarrow$ x86 instruction level SE

KLEE, C.Cadar et al.

$\Rightarrow$ LLVM bytecode level SE

It is now a question of applying it to vulnerability analysis

Motivating Example

```c
#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;
}
```

```c
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");
  }
}
```

Motivating Example

```c
#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;
}
```

```c
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");
  }
}
```

Goal

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

Motivating Example

```c
#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;
}
```

```c
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");
  }
}
```

$\exists i.\exists s.\exists m_0.\exists p_0.$

i: input $\quad m$: memory

s: secret $\quad p$: stack pointer

$$p_1 \triangleq p_0 - \texttt{SIZE}$$
$$p_2 \triangleq p_1 - \texttt{SIZE}$$
$$m_1 \triangleq m_0\,[p_1 \,..\, p_1 + \texttt{SIZE} - 1] \leftarrow s$$
$$m_2 \triangleq m_1\,[p_2 \,..\, p_2 + N - 1] \leftarrow i$$

$$m_2\,[p_1 \,..\, p_1 + \texttt{SIZE} - 1] = m_2\,[p_2 \,..\, p_2 + \texttt{SIZE} - 1]$$

Binary-Level Semantics

$$
\begin{array}{l}
\exists i.\exists s.\exists m_0.\exists p_0. \\
\quad p_1 \triangleq p_0 - \texttt{SIZE} \\
\quad p_2 \triangleq p_1 - \texttt{SIZE} \\
\quad m_1 \triangleq m_0\,[p_1 \,..\, p_1 + \texttt{SIZE} - 1] \leftarrow s \\
\quad m_2 \triangleq m_1\,[p_2 \,..\, p_2 + N - 1] \leftarrow i \\
m_2\,[p_1 \,..\, p_1 + \texttt{SIZE} - 1] = m_2\,[p_2 \,..\, p_2 + \texttt{SIZE} - 1]
\end{array}
$$

oversimplified formula!

The real formula is about 2130 reads and 456 writes

Binary-Level Semantics

$$
\begin{aligned}
&\exists i.\exists s.\exists m_0.\exists p_0.\\
&\quad p_1 \triangleq p_0 - \texttt{SIZE}\\
&\quad p_2 \triangleq p_1 - \texttt{SIZE}\\
&\quad m_1 \triangleq m_0\,[p_1 \,..\, p_1 + \texttt{SIZE} - 1] \leftarrow s\\
&\quad m_2 \triangleq m_1\,[p_2 \,..\, p_2 + N - 1] \leftarrow i\\
&m_2\,[p_1 \,..\, p_1 + \texttt{SIZE} - 1] = m_2\,[p_2 \,..\, p_2 + \texttt{SIZE} - 1]
\end{aligned}
$$

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 hours on these formulas

ASPack case study: 293 000 reads, 58 000 writes

$\Rightarrow$ 24 hours of resolution !

Security Is Not Safety

Sending the formula to a solver:

$$\Rightarrow \left\{ s_{[0\,..\,\texttt{SIZE}-1]} = 0, i_{[0\,..\,\texttt{SIZE}-1]} = 0, \ldots \right\}$$

"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

Security Is Not Safety

Sending the formula to a solver:

$\Rightarrow \left\{ s_{[0\,..\,\texttt{SIZE}-1]} = 0, i_{[0\,..\,\texttt{SIZE}-1]} = 0, \ldots \right\}$

"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

Section 1

Model Generation for Quantified Formulas: A Taint-Based Approach

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]

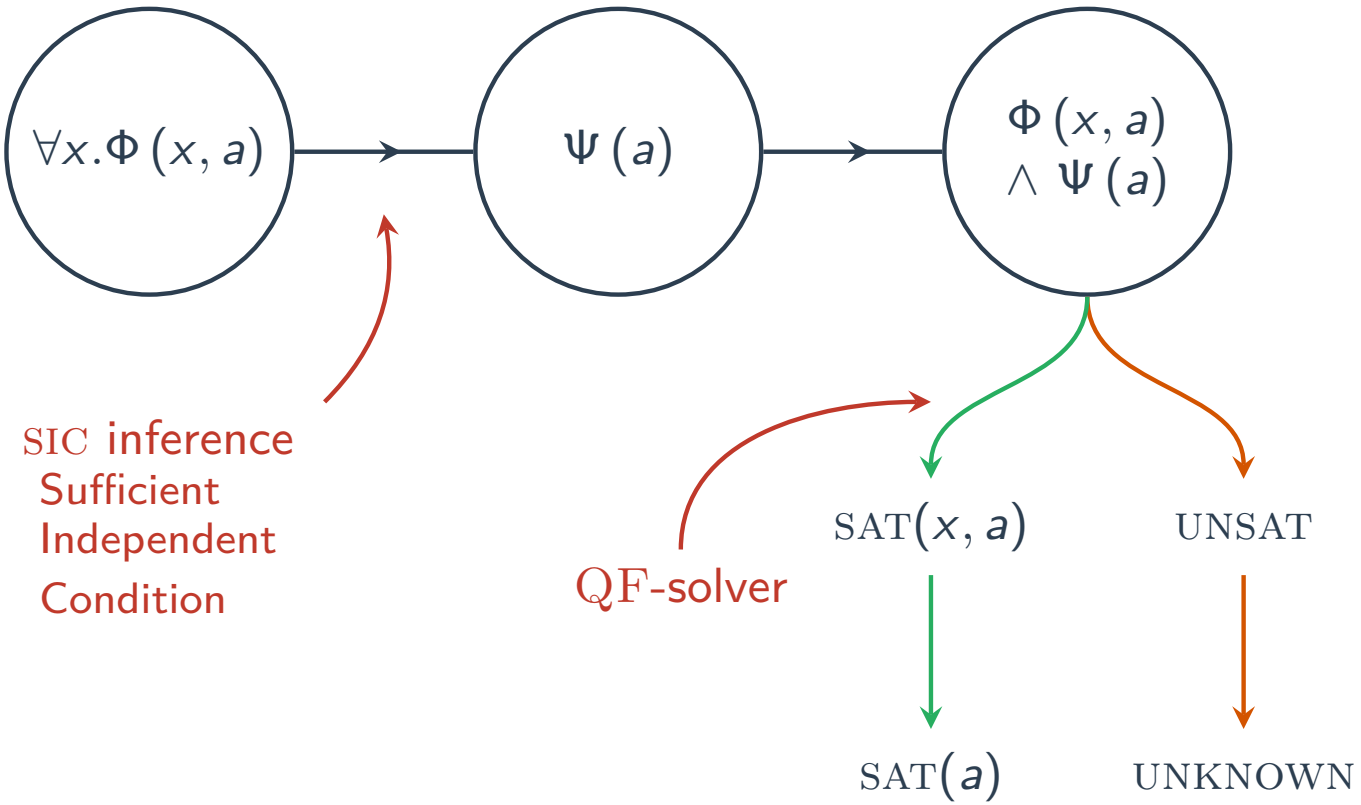
Toy Example

```
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: $a = 1 \wedge x = 1 \wedge b = 0$
  - A good solution: $a = 0 \wedge x = 1 \wedge b = 1$
- The constraint $a = 0$ is the independence condition
- Quantifier-free reachability condition: $(ax + b > 0) \wedge (a = 0)$

Our Proposal in a Nutshell



Sufficient Independence Condition (SIC)

Sufficient Independence Condition (SIC)

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

formula indep.

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

Model generalization

- Let $\Phi(\boldsymbol{x}, \boldsymbol{a})$ a formula and $\Psi(\boldsymbol{a})$ a $\text{SIC}_{\Phi,\boldsymbol{x}}$.
- If there exists an interpretation $\{\mathtt{x}, \mathtt{a}\}$ such that $\{\mathtt{x}, \mathtt{a}\} \models \Psi(\boldsymbol{a}) \wedge \Phi(\boldsymbol{x}, \boldsymbol{a})$, then $\{\mathtt{a}\} \models \forall \boldsymbol{x}.\Phi(\boldsymbol{x}, \boldsymbol{a})$.

Weakest Independence Condition (WIC)

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

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

Model specialization

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

Taint-based SIC inference

Function `inferSIC(Φ,x)`:

Input: Φ a formula and x a set of targeted variables

Output: Ψ a $\mathrm{SIC}_{\Phi,x}$

either Φ *is a constant*

return $\top$

either Φ *is a variable* v

return $v \notin x$

either Φ *is a function* $f(\phi_1,..,\phi_n)$

Let $\psi_i \triangleq \texttt{inferSIC}(\phi_i, x)$ for all $i \in \{1,..,n\}$

Let $\Psi \triangleq \texttt{theorySIC}(f, (\phi_1,..,\phi_n), (\psi_1,..,\psi_n), x)$

return $\Psi \vee \bigwedge_i \psi_i$

syntactic part

a and b indep$_x$ $\rightsquigarrow$ $f(a,b)$ indep$_x$

semantic part

a indep$_x$ and $a = 0$ $\rightsquigarrow$ $a \cdot * $ indep$_x$

Taint-based SIC inference

Proposition

- If $\texttt{theorySIC}(f, \phi_i, \psi_i, \boldsymbol{x})$ computes a $\mathrm{SIC}_{f(\phi_i),\boldsymbol{x}}$, then $\texttt{inferSIC}(\Phi, \boldsymbol{x})$ computes a $\mathrm{SIC}_{\Phi,\boldsymbol{x}}$.

Function `inferSIC(`Φ`,`$\boldsymbol{x}$`):`

Input: Φ a formula and $\boldsymbol{x}$ a set of targeted variables

Output: Ψ a $\mathrm{SIC}_{\Phi,\boldsymbol{x}}$

either Φ *is a constant*
  return $\top$

either Φ *is a variable* v — syntactic part
  return $v \notin \boldsymbol{x}$
  a and b indep$_{\boldsymbol{x}}$ $\rightsquigarrow$ $f\,(a, b)$ indep$_{\boldsymbol{x}}$

either Φ *is a function* $f\,(\phi_1, .., \phi_n)$
  Let $\psi_i \triangleq \texttt{inferSIC}\,(\phi_i, \boldsymbol{x})$ for all $i \in \{1, .., n\}$
  Let $\Psi \triangleq \texttt{theorySIC}\,(f, (\phi_1, .., \phi_n), (\psi_1, .., \psi_n), \boldsymbol{x})$
  return $\Psi \vee \bigwedge_i \psi_i$ — semantic part
  a indep$_{\boldsymbol{x}}$ and $a = 0$ $\rightsquigarrow$ $a \cdot * $ indep$_{\boldsymbol{x}}$

theorySIC defined as a recursive function

$$
\begin{aligned}
(a \Rightarrow b)^\bullet &\triangleq (a^\bullet \wedge a = \bot) \vee (b^\bullet \wedge b = \top) \\
(a \wedge b)^\bullet &\triangleq (a^\bullet \wedge a = \bot) \vee (b^\bullet \wedge b = \bot) \\
(a \vee b)^\bullet &\triangleq (a^\bullet \wedge a = \top) \vee (b^\bullet \wedge b = \top) \\
(\text{ite}\ c\ a\ b)^\bullet &\triangleq (c^\bullet \wedge \text{ite}\ c\ a^\bullet\ b^\bullet) \vee (a^\bullet \wedge b^\bullet \wedge a = b)
\end{aligned}
$$

$$
\begin{aligned}
(a_n \wedge b_n)^\bullet &\triangleq (a_n^\bullet \wedge a_n = 0_n) \vee (b_n^\bullet \wedge b_n = 0_n) \\
(a_n \vee b_n)^\bullet &\triangleq (a_n^\bullet \wedge a_n = 1_n) \vee (b_n^\bullet \wedge b_n = 1_n) \\
(a_n \times b_n)^\bullet &\triangleq (a_n^\bullet \wedge a_n = 0_n) \vee (b_n^\bullet \wedge b_n = 0_n) \\
(a_n \ll b_n)^\bullet &\triangleq (b_n^\bullet \wedge b_n \geq n)
\end{aligned}
$$

$$
\begin{aligned}
((a\,[i] \leftarrow e)\,[j])^\bullet &\triangleq (\text{ite}\ (i = j)\ e\ (a\,[j]))^\bullet \\
&\triangleq ((i = j)^\bullet \wedge (\text{ite}\ (i = j)\ e^\bullet\ (a\,[j])^\bullet)) \\
&\quad \vee (e^\bullet \wedge (a\,[j])^\bullet \wedge (e = a\,[j])) \\
&\triangleq (i^\bullet \wedge j^\bullet \wedge (\text{ite}\ (i = j)\ e^\bullet\ (a\,[j])^\bullet)) \\
&\quad \vee (e^\bullet \wedge (a\,[j])^\bullet \wedge (e = a\,[j]))
\end{aligned}
$$

Experimental Evaluation

Boolector: an efficient QF-solver for bitvectors and arrays

Best approaches

		Z3	Btor•	Btor• ▷ Z3
SMT-LIB	SAT	261	399	485
	# UNSAT	165	N/A	165
	UNKNOWN	843	870	619
	total time	270 150	350	94 610
BINSEC	SAT	953	1042	1067
	# UNSAT	319	N/A	319
	UNKNOWN	149	379	35
	total time	64 761	1 152	1 169

GRUB example

	Z3	Btor•
SAT	1	**540**
# UNSAT	42	N/A
UNKNOWN	852	355
total time	159 765	**16 732**

Complementarity with existing solvers (SAT instances)

		CVC4•	Z3•	Btor•
SMT-LIB	CVC4	-10 +168 [252]		-10 +325 [409]
	Z3		-119 +224 [485]	-86 +224 [485]
BINSEC	CVC4	-25 +28 [979]		-25 +116 [1067]
	Z3		-25 +114 [1067]	-25 +114 [1067]

solver•: solver enhanced with our method

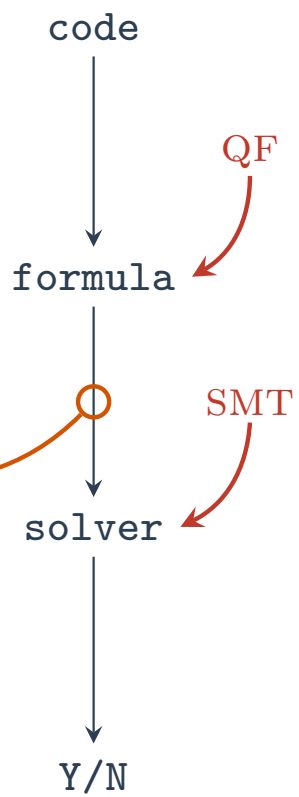
Section 2

Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing

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

  $\Rightarrow$ 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]

Array Theory

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 $i \in \mathcal{I}$: $a[i] \leftarrow e$

$$\cdot[\cdot] : \text{Array } \mathcal{I}\ \mathcal{E} \to \mathcal{I} \to \mathcal{E}$$
$$\cdot[\cdot] \leftarrow \cdot : \text{Array } \mathcal{I}\ \mathcal{E} \to \mathcal{I} \to \mathcal{E} \to \text{Array } \mathcal{I}\ \mathcal{E}$$

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

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

Arrays in Practice

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)!

ROW Simplification

An example coming from binary analysis

$\texttt{esp}_0 : \texttt{BitVec16}$
$\texttt{mem}_0 : \texttt{Array BitVec16 BitVec16}$

$\texttt{assert}\,(\texttt{esp}_0 > 61440)$
$\texttt{mem}_1 \triangleq \texttt{mem}_0\,[\texttt{esp}_0 - 16] \leftarrow 1415$
$\texttt{esp}_1 \triangleq \texttt{esp}_0 - 64$
$\texttt{eax}_0 \triangleq \texttt{mem}_1\,[\texttt{esp}_1 + 48]$
$\texttt{assert}\,(\texttt{mem}_1\,[\texttt{eax}_0] = 9265)$

$\rightsquigarrow$

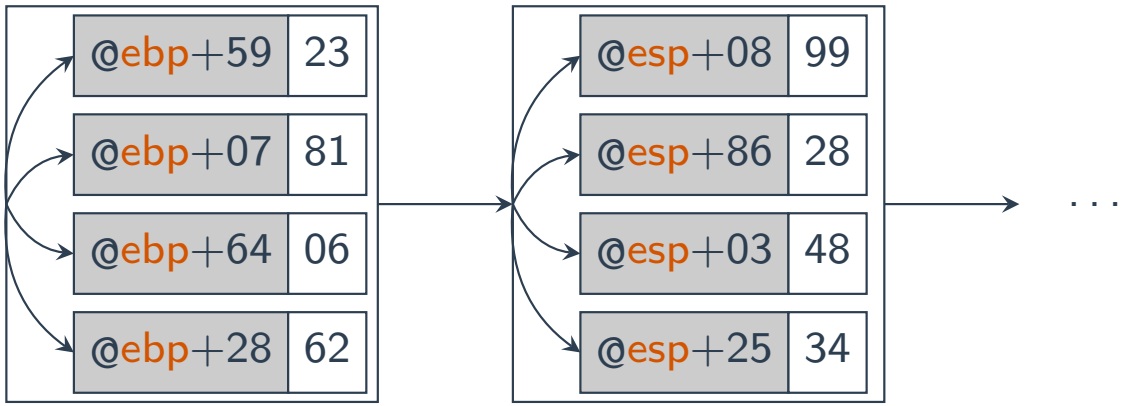
$\texttt{esp}_0 : \texttt{BitVec16}$
$\texttt{mem}_0 : \texttt{Array BitVec16 BitVec16}$

$\texttt{assert}\,(\texttt{esp}_0 > 61440)$
$\texttt{assert}\,(\texttt{mem}_0\,[1415] = 9265)$

These simplifications depend on two factors

- The equality check procedure
  verify that $\texttt{esp}_1 + 48 = \texttt{esp}_0 - 16$
  $\Rightarrow$ **precise reasoning**: base normalization + abstract domains
- The underlying representation of an array
  remember that $\texttt{mem}_1\,[\texttt{esp}_1 + 48] = 1415$
  $\Rightarrow$ **scalability** issue: list-map representation

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

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...

$\Rightarrow$ Reduce the number of bases

Associate to every indices i an abstract domain $i^\sharp$

- If $i^\sharp \sqcap j^\sharp = \bot$ then $(a\,[i] \leftarrow e)\,[j] = a\,[j]$
- Integrated in the list-map representation

$\Rightarrow$ Prove disequality between different bases

Impact of the Simplification: Medium-Size Formulas

- 6,590 x 3 medium-size formulas from static SE
- TIMEOUT = 1,000 seconds

		simpl. time	#TIMEOUT and resolution time						#ROW
			Boolector		Yices		Z3		
concrete	default	61	0	163	2	69	0	872	**866,155**
	FAS	85	0	94	2	68	0	244	**1,318**
	FAS-itv	111	0	94	2	68	0	224	1,318
interval	default	65	0	**2,584**	2	465	31	155,992	866,155
	FAS	99	0	2,245	2	487	25	126,806	531,654
	FAS-itv	118	0	**755**	2	140	14	37,269	205,733
symbolic	default	61	0	6,173	3	1,961	**65**	305,619	866,155
	FAS	91	0	6,117	3	1,965	66	158,635	531,654
	FAS-itv	111	0	4,767	2	1,108	**43**	80,569	295,333

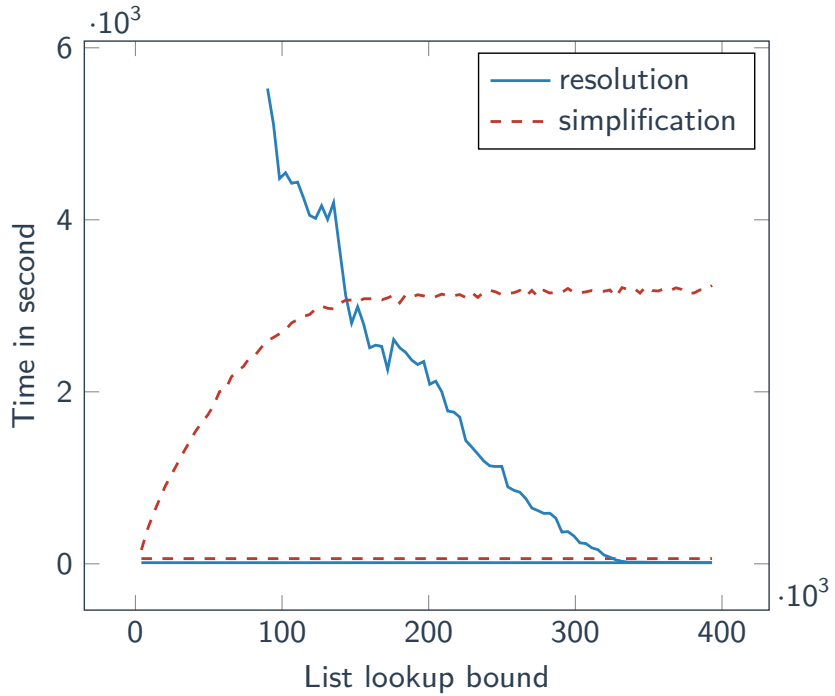
Impact of the Simplification: Very Large Formulas

- 29 x 3 very large formulas from dynamic SE
- TIMEOUT = 1,000 seconds

		simpl. time	#TIMEOUT and resolution time						#ROW
			Boolector		Yices		Z3		
concrete	default	44	10	159	4	1,098	26	3.33	**1,120,798**
	FAS-list	**1,108**	8	845	4	198	10	918	456,915
	FAS	196	8	820	4	196	10	922	456,915
	FAS-itv	210	4	654	1	12	4	1,120	**0**
interval	default	44	12	131	12	**596**	**27**	0.19	1,120,798
	FAS-list	222	12	129	12	595	26	236	657,594
	FAS	231	12	129	12	597	26	291	657,594
	FAS-itv	237	12	58	12	**28**	**19**	81	651,449
symbolic	default	40	12	1,522	12	1,961	27	0.13	1,120,798
	FAS-list	187	11	1,199	12	2,018	26	486	657,594
	FAS	194	11	1,212	12	2,081	26	481	657,594
	FAS-itv	200	11	1,205	12	2,063	26	416	657,594

Focus on Specific Case: the ASPack Example

- 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

Overview

- 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]

Motivating Example Remembered

```c
#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;
}
```

```c
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");
  }
}
```

$\exists i.\exists s.\exists m_0.\exists p_0.$

i: input $\quad m$: memory

s: secret $\quad p$: stack pointer

$$p_1 \triangleq p_0 - \texttt{SIZE}$$
$$p_2 \triangleq p_1 - \texttt{SIZE}$$
$$m_1 \triangleq m_0\,[p_1 \,..\, p_1 + \texttt{SIZE} - 1] \leftarrow s$$
$$m_2 \triangleq m_1\,[p_2 \,..\, p_2 + N - 1] \leftarrow i$$

$$m_2\,[p_1 \,..\, p_1 + \texttt{SIZE} - 1] = m_2\,[p_2 \,..\, p_2 + \texttt{SIZE} - 1]$$

Robust Symbolic Execution in Practice

$$
\begin{array}{l}
\exists i.\exists s.\exists m_0.\exists p_0. \\
\quad \begin{array}{rcl} p_1 & \triangleq & p_0 - \texttt{SIZE} \\ p_2 & \triangleq & p_1 - \texttt{SIZE} \\ m_1 & \triangleq & m_0\,[p_1 \,..\, p_1 + \texttt{SIZE} - 1] \leftarrow s \\ m_2 & \triangleq & m_1\,[p_2 \,..\, p_2 + N - 1] \leftarrow i \end{array} \\
m_2\,[p_1 \,..\, p_1 + \texttt{SIZE} - 1] = m_2\,[p_2 \,..\, p_2 + \texttt{SIZE} - 1]
\end{array}
$$

Sending the formula to a solver:

$\Rightarrow \left\{ s_{[0\,..\,\texttt{SIZE}-1]} = 0, i_{[0\,..\,\texttt{SIZE}-1]} = 0, \ldots \right\}$

- 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

Robust Symbolic Execution in Practice

$$
\begin{array}{l}
\exists i.\forall s.\forall m_0.\forall p_0. \\
\quad \begin{array}{rcl}
p_1 & \triangleq & p_0 - \texttt{SIZE} \\
p_2 & \triangleq & p_1 - \texttt{SIZE} \\
m_1 & \triangleq & m_0\,[p_1 \,..\, p_1 + \texttt{SIZE} - 1] \leftarrow s \\
m_2 & \triangleq & m_1\,[p_2 \,..\, p_2 + N - 1] \leftarrow i
\end{array} \\
m_2\,[p_1 \,..\, p_1 + \texttt{SIZE} - 1] = m_2\,[p_2 \,..\, p_2 + \texttt{SIZE} - 1]
\end{array}
$$

Problems:

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

$\Rightarrow$ The Death of SMT Solvers

Robust Symbolic Execution in Practice

$$\exists i.\ \exists s.\ \exists m_0.\ \forall p_0.$$
$$p_1 \triangleq p_0 - \texttt{SIZE}$$
$$p_2 \triangleq p_1 - \texttt{SIZE}$$
$$m_1 \triangleq m_0\,[p_0 - \texttt{SIZE}\,..\,p_0 - 1] \leftarrow s$$
$$m_2 \triangleq m_1\,[p_0 - 2 \cdot \texttt{SIZE}\,..\,p_0 - 2 \cdot \texttt{SIZE} + N - 1] \leftarrow i$$
$$i[0\,..\,\texttt{SIZE} - 1] = i[\texttt{SIZE}\,..\,2 \cdot \texttt{SIZE} - 1]$$
$$\wedge\ N \geq 2 \cdot \texttt{SIZE}$$

Problems:

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

$\Rightarrow$ ~~The Death of SMT Solvers~~

For example with $\texttt{SIZE} = 8$,

- input `abcdefghabcdefgh` leads to the "Success!" branch
- buffer overflow in `read_input`

Experimental Evaluation

- set of `crackme` challenges
- compare true and false positives

	SE robust		
	true positives	false positives	UNKNOWN
Boolector	N/A	N/A	24
CVC4	5	**0**	19
Yices	N/A	N/A	24
Z3	7	**0**	17

	SE classic		
	true positives	false positives	UNKNOWN
Boolector	**12**	11	1
CVC4	7	9	8
Yices	7	11	6
Z3	**12**	12	0

	SE robust + elim.		
	true positives	false positives	UNKNOWN
Boolector	**12**	**0**	12
CVC4	7	**0**	17
Yices	7	**0**	17
Z3	**12**	**0**	12

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

Summary

- 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

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

1. Model Generation for Quantified Formulas
   - More precise inference mechanisms of independence conditions
   - Identification of subclasses for which inferring WIC is feasible
   - Combination with other quantifier instantiation techniques
2. Arrays Made Simpler
   - Deeper integration inside a dedicated array solver
   - Adding more expressive domain reasoning
3. Robust Symbolic Execution
   - Precise evaluation of our semi-automatic incremental specification procedure
   - Thorough comparison of Robust Symbolic Execution to other techniques
4. Beyond that
   - Restrict to "there exists" and "for all" quantifications is not nuanced enough
   - Might want to say that an event "almost always" or "almost never" occurs

   $\Rightarrow$ Requires notions of probabilities or model counting

Bibliography

Benjamin Farinier, S. Bardin, R. Bonichon, and M. Potet.

Model generation for quantified formulas: A taint-based approach.

In *Computer Aided Verification - 30th International Conference, CAV 2018, Oxford, UK, July 14-17, 2018*, 2018.

Benjamin Farinier, R. David, S. Bardin, and M. Lemerre.

Arrays made simpler: An efficient, scalable and thorough preprocessing.

In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, November 16-21, 2018*, 2018.

Benjamin Farinier, S. Bardin, R. Bonichon, and M. Potet.

Génération de modèles pour les formules quantifiées : une approche basée sur la teinte.

In *Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL 2018, Grenoble, France, June 13-15, 2018*, 2018.

Benjamin Farinier, R. David, and S. Bardin.

Simplification efficace pour la théorie des tableaux.

In *Journées Francophones des Langages Applicatifs, JFLA 2018, Banyuls-sur-Mer, France, January 24-27, 2018*, 2018.

Benjamin Farinier, S. Bardin, R. Bonichon, and M. Potet.

En finir avec les faux positifs grâce à l'exécution symbolique robuste.

In *Journées Francophones des Langages Applicatifs, JFLA 2019, Les Rousses, France, January 30-February 2, 2019*, 2019.