Static Analysis for Security Properties of Software by Abstract Interpretation

Francesco Parolini

PhD Defense
APR team
LIP6, Sorbonne Université
Paris, France

26/06/2024

~ Introduction ~

~ Software & errors

Software is everywhere

~ Software & errors

Software is everywhere

Increasing size and complexity $\implies$ more bugs

~ Software & errors

Software is everywhere

Increasing size and complexity $\Longrightarrow$ more bugs

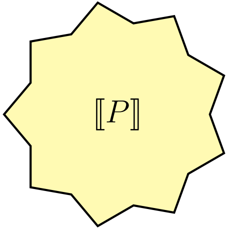
~ Software & errors

Software is everywhere

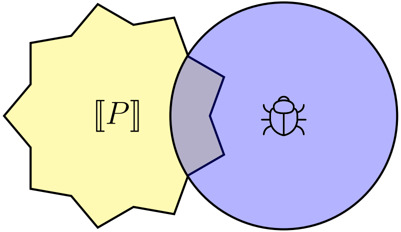
Increasing size and complexity $\Longrightarrow$ more bugs

We need techniques for **reliable software**

~ Software verification

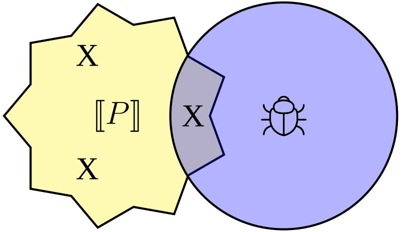


~ Software verification



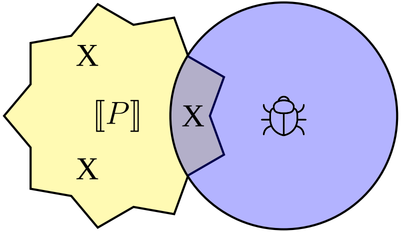
$[\![P]\!] \subseteq^{?}$ Safe

~ Software verification



$[\![P]\!] \subseteq^? \text{Safe}$

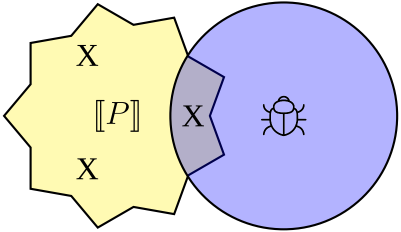
~ Software verification



$[\![P]\!] \subseteq^? \text{Safe}$

“Absence of evidence is not evidence of absence”

~ Software verification



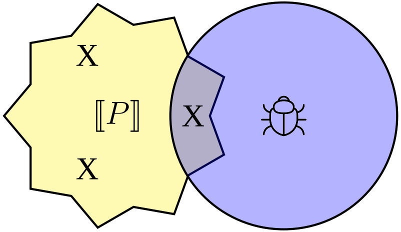
$[\![P]\!] \subseteq^{?}$ Safe

"Absence of evidence is not evidence of absence"

Theorem (Rice)

All non-trivial program properties are ***not computable***

~ Software verification



$[\![P]\!] \subseteq^? \text{Safe}$

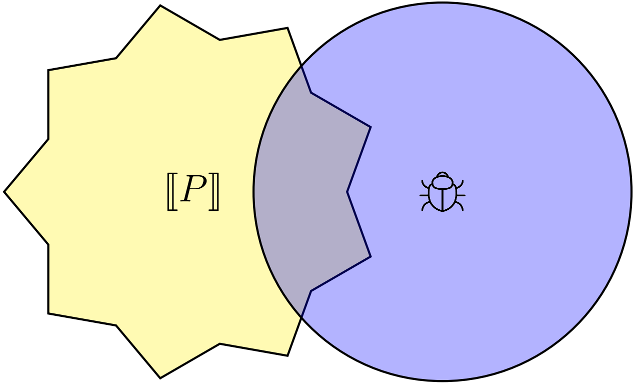
"Absence of evidence is not evidence of absence"

Theorem (Rice)

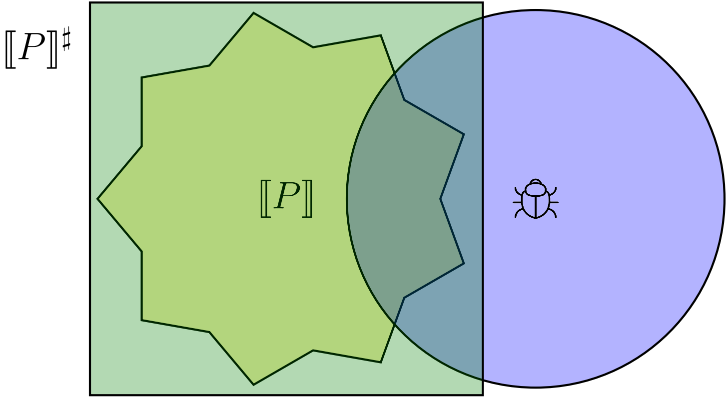
All non-trivial program properties are ***not computable***

Formal methods study **trade-offs** to prove correctness

~ Abstract interpretation

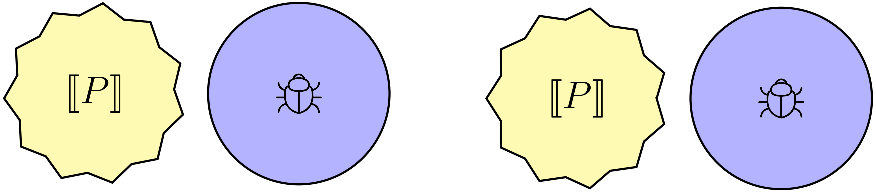


~ Abstract interpretation

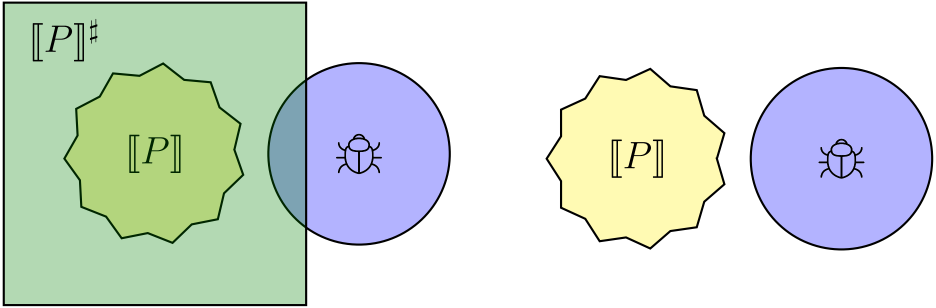


$$[\![P]\!] \subseteq [\![P]\!]^\sharp \subseteq^? \text{Safe}$$

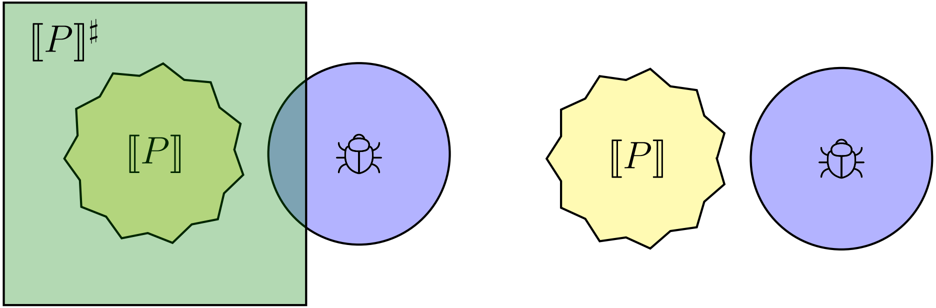
~ False positives and negatives



~ False positives and negatives

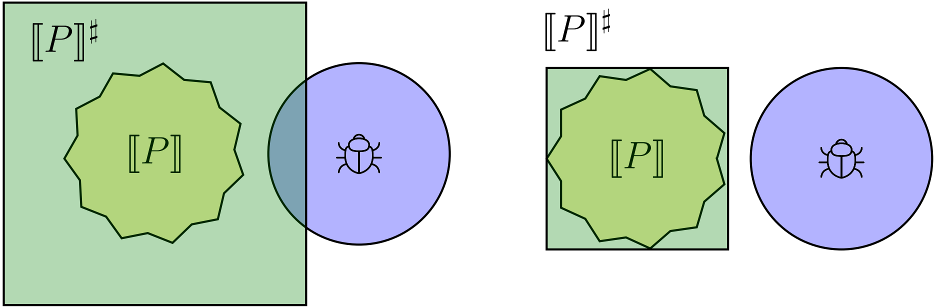


~ False positives and negatives



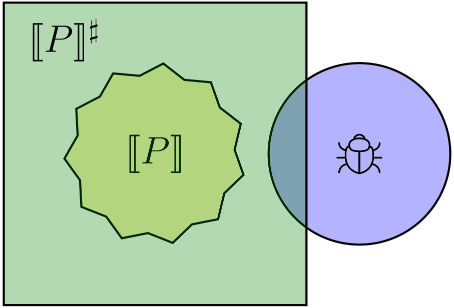
Can raise **false positives**

~ False positives and negatives

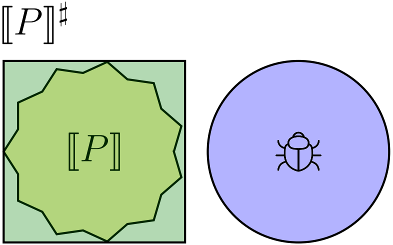


Can raise **false positives**

~ False positives and negatives



Can raise **false positives**



Forbids **false negatives**

~ The case of cybersecurity

Security vulnerabilities matter to

- Citizens

Social Media Hacking: Statistics Overview

Cybercrimes on social media platforms account for $3.25 Billion in annual global revenue.

This statistic demonstrates the magnitude of the problem. The $3.25 billion in annual global revenue lost to cybercrimes on social media highlights the need for increased security measures to protect users from malicious actors. It also underscores the importance of educating users on how to protect themselves from cyberattacks.

~ The case of cybersecurity

Security vulnerabilities matter to

- Citizens
- Companies

Microsoft

NEWS

Microsoft got hacked by state sponsored group it was investigating

Posted: January 23, 2024 by Pieter Arntz

In a spy-vs-spy type of scenario, Microsoft has acknowledged that a group called Midnight Blizzard (also known as APT29 or Cozy Bear), gained access to a Microsoft legacy non-production test tenant account.

~ The case of cybersecurity

Security vulnerabilities matter to

- Citizens
- Companies
- Governments

News / Canadian Politics / Canada

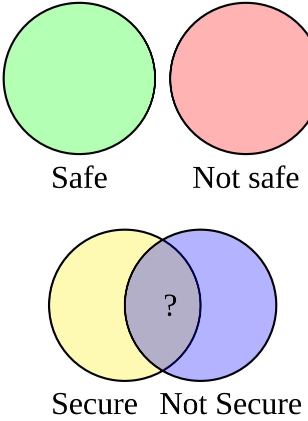
Global Affairs investigating 'malicious' hack after VPN compromised for over one month

A month-long cyber breach forced the department to shut down some internal services and appears to have compromised the data and emails of numerous employees

Christopher Nardi

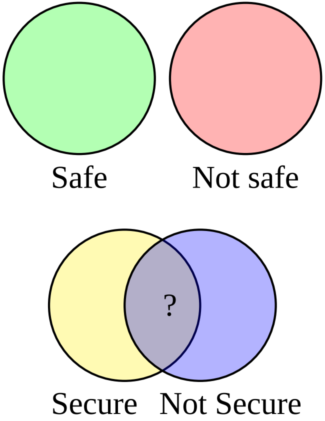
Published Jan 30, 2024 • Last updated Jan 30, 2024 • 3 minute read

~ The challenges of cybersecurity



Defining when a program is **secure**

~ The challenges of cybersecurity

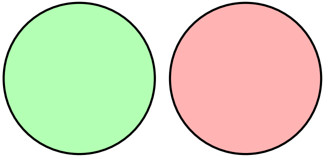


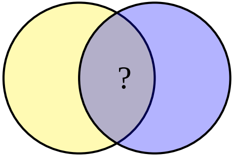
Defining when a program is **secure**

Security spans over

- Availability
- Confidentiality
- Integrity

~ The challenges of cybersecurity





Defining when a program is **secure**

Security spans over

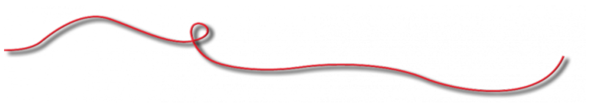
- Availability
- Confidentiality
- Integrity

Adapt existing techniques to security

~ This thesis

Techniques to prove programs secure by **formal reasoning** for **ReDoS attacks** and **exploitable runtime errors**

~ This thesis



Techniques to prove programs secure by **formal reasoning** for **ReDoS attacks** and **exploitable runtime errors**

Semantic frameworks

~ This thesis

Techniques to prove programs secure by **formal reasoning**
for **ReDoS attacks** and **exploitable runtime errors**

Semantic frameworks

Mathematical formalization of the vulnerabilities

~ This thesis

Techniques to prove programs secure by **formal reasoning**
for **ReDoS attacks** and **exploitable runtime errors**

Semantic frameworks

Mathematical formalization of the vulnerabilities

Sound, automatic analyses

~ This thesis

Techniques to prove programs secure by **formal reasoning** for **ReDoS attacks** and **exploitable runtime errors**

Semantic frameworks

Mathematical formalization of the vulnerabilities

Sound, automatic analyses

Experiments on real-world data

~ Part I: Regular Expression Denial of Service Attacks ~

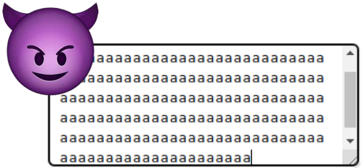
~ Introduction ~

~ ReDoS attacks: what

Regular expression **D**enial **o**f **S**ervice (ReDoS)

Algorithmic complexity attack

Matching engines have **exponential complexity**

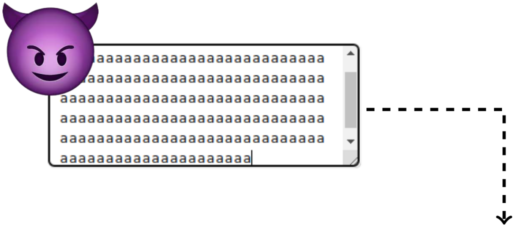


~ ReDoS attacks: what

Regular expression **D**enial **o**f **S**ervice (ReDoS)

Algorithmic complexity attack

Matching engines have **exponential complexity**



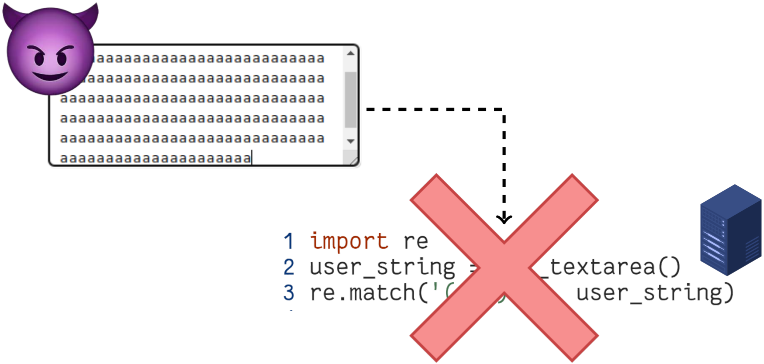
```python
import re
user_string = get_textarea()
re.match('(a|a)*b', user_string)
```

~ ReDoS attacks: what

Regular expression **D**enial **o**f **S**ervice (ReDoS)

Algorithmic complexity attack

Matching engines have **exponential complexity**

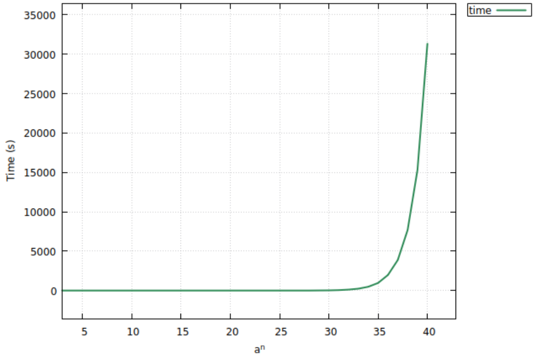


~ Example

```python
import re

email_regex = \
    r'^([0-9a-zA-Z]([-.\w]*[0-9a-zA-Z])*\
      @(([0-9a-zA-Z])+([-\w]*[0-9a-zA-Z])*\.)+[a-zA-Z]{2,9})$'

for i in range(1, 41):
    re.match(email_regex, 'a' * i)
```



~ ReDoS attacks: some numbers

42% of the 4,000 mostly starred Python projects on Github **use regexes**

~ ReDoS attacks: some numbers

42% of the 4,000 mostly starred Python projects on Github **use regexes**

10% of the Node.js-based webservers are vulnerable

~ ReDoS attacks: some numbers

42% of the 4,000 mostly starred Python projects on Github **use regexes**

10% of the Node.js-based webservers are vulnerable

Vulnerable languages

~ ReDoS attacks: some numbers

42% of the 4,000 mostly starred Python projects on Github **use regexes**

10% of the Node.js-based webservers are vulnerable

Vulnerable languages

Only **38%** of the developers know about the **existence** of ReDoS

~ Real-world consequences of ReDoS

Stack Exchange Network Status

This account is no longer active. For updates on status, head to https://www.stackstatus.net/

Outage Postmortem - July 20, 2016

Overview

On July 20, 2016 we experienced a 34 minute outage starting at 14:44 UTC. It took 10 minutes to identify the cause, 14 minutes to write the code to fix it, and 10 minutes to roll out the fix to a point where Stack Overflow became available again.

The direct cause was a malformed post that caused one of our regular expressions to consume high CPU on our web servers. The post was in the homepage list, and that caused the expensive regular expression to be called on each home page view. This caused the home page to stop responding fast enough. Since the home page is what our load balancer uses for the health check, the entire site became unavailable since the load balancer took the servers out of rotation.

EXPLOIT DATABASE

Apple Safari 2.0.4 - JavaScript Regular Expression Match Remote Denial of Service

EDB-ID: 29007 | CVE: 2006-6015 | Author: JBH_CG | Type: DOS | Platform: OSX | Date: 2006-11-14

EDB Verified: ✓ | Exploit: / {} | Vulnerable App:

```
source: https://www.securityfocus.com/bid/21053/info

Apple Safari web browser is prone to a denial-of-service vulnerability when executing certain JavaScript code.

An attacker can exploit this issue to crash an affected browser. Presumably, this issue may also result in remote code execution, but this has not been confirmed.

Apple Safari 2.0.4 is vulnerable to this issue; other versions may also be affected.

<script> var reg = /(.)*/; var z = 'Z'; while (z.length <= 8192) z+=z; var boum = reg.exec(z); </script>
```

Get Started Free | Contact Sales: +1 (888) 993-5273 |

CLOUDFLARE The Cloudflare Blog

Subscribe to receive notifications of new posts: Email Address Subscribe

All Posts | Product News | Speed & Reliability | Security | Zero Trust | Developers | AI | Policy | Partners | Life at Cloudflare

Details of the Cloudflare outage on July 2, 2019

07/12/2019

The events of July 2

On July 2, we deployed a new rule in our WAF Managed Rules that caused CPUs to become exhausted on every CPU core that handles HTTP/HTTPS traffic on the Cloudflare network worldwide. We are constantly improving WAF Managed Rules to respond to new vulnerabilities and threats. In May, for example, we used the speed with which we can update the WAF to push a rule to protect against a serious SharePoint vulnerability. Being able to deploy rules quickly and globally is a critical feature of our WAF.

Unfortunately, last Tuesday's update contained a regular expression that backtracked enormously and exhausted CPU used for HTTP/HTTPS serving. This brought down Cloudflare's core proxying, CDN and WAF functionality. The following graph shows CPUs dedicated to serving HTTP/HTTPS traffic spiking to nearly 100% usage across the servers in our network.

Version 2.5.1

January 6, 2021

CairoSVG 2.5.1 has been released!

WARNING: this is a security update.

When processing SVG files, CairoSVG was using two regular expressions which are vulnerable to Regular Expression Denial of Service (REDoS).

If an attacker provided a malicious SVG, it could make CairoSVG get stuck processing the file for a very long time.

Other bug fixes:

- Fix marker positions for unclosed paths
- Follow hint when only output_width or output_height is set
- Handle opacity on raster images
- Don't crash when use tags reference unknown tags
- Take care of the next letter when A/a is replaced by l
- Fix misalignment in node.vertices

~ Static analysis of ReDoS

Framework for **static ReDoS detection**

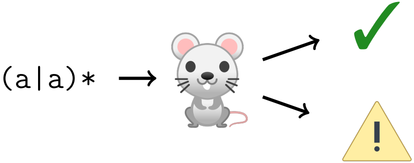
- A tree semantics for the matching
- Sound, fast, and precise analysis

~ Static analysis of ReDoS

Framework for **static ReDoS detection**

- A tree semantics for the matching
- Sound, fast, and precise analysis

Implemented it in the `rat` (**R**eDoS **A**bstract **T**ester) tool

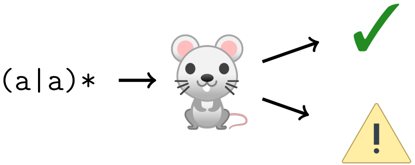


~ Static analysis of ReDoS

Framework for **static ReDoS detection**

- A tree semantics for the matching
- Sound, fast, and precise analysis

Implemented it in the `rat` (**R**eDoS **A**bstract **T**ester) tool



Compared to seven other ReDoS detectors

~ Semantics ~

~ Semantics of regex matching

Let $R \in Regex$ and $w \in Words$. We define $[\![R]\!](w)$ as a tree.

$$\langle (a \mid a)^*, ab \rangle$$

~ Semantics of regex matching

Let $R \in Regex$ and $w \in Words$. We define $[\![R]\!](w)$ as a tree.

$$\langle (a \mid a)^*, ab \rangle$$

$$\langle (a \mid a)(a \mid a)^{\overline{*}}, ab \rangle$$

~ Semantics of regex matching

Let $R \in Regex$ and $w \in Words$. We define $[\![R]\!](w)$ as a tree.

$$\langle (a \mid a)^*, ab \rangle$$

$$\langle (a \mid a)(a \mid a)^{\overline{*}}, ab \rangle$$

$$\langle a(a \mid a)^{\overline{*}}, ab \rangle$$

~ Semantics of regex matching

Let $R \in Regex$ and $w \in Words$. We define $[\![R]\!](w)$ as a tree.

$$\langle (a \mid a)^*, ab \rangle$$

$$\langle (a \mid a)(a \mid a)^{\overline{*}}, ab \rangle$$

$$\langle a(a \mid a)^{\overline{*}}, ab \rangle$$

$$\langle (a \mid a)^*, b \rangle$$

~ Semantics of regex matching

Let $R \in Regex$ and $w \in Words$. We define $[\![R]\!](w)$ as a tree.

$$\langle (a \mid a)^*, ab \rangle$$

$$\langle (a \mid a)(a \mid a)^{\overline{*}}, ab \rangle$$

$$\langle a(a \mid a)^{\overline{*}}, ab \rangle$$

$$\langle (a \mid a)^*, b \rangle$$

$$\langle (a \mid a)(a \mid a)^{\overline{*}}, b \rangle$$

~ Semantics of regex matching

Let $R \in Regex$ and $w \in Words$. We define $[\![R]\!](w)$ as a tree.

$$\langle (a \mid a)^*, ab \rangle$$

$$\langle (a \mid a)(a \mid a)^{\overline{*}}, ab \rangle$$

$$\langle a(a \mid a)^{\overline{*}}, ab \rangle$$

$$\langle (a \mid a)^*, b \rangle$$

$$\langle (a \mid a)(a \mid a)^{\overline{*}}, b \rangle$$

$$\langle a(a \mid a)^{\overline{*}}, b \rangle$$

~ Semantics of regex matching

Let $R \in Regex$ and $w \in Words$. We define $[\![R]\!](w)$ as a tree.

$$\langle (a \mid a)^*, ab \rangle$$

$$\langle (a \mid a)(a \mid a)^{\overline{*}}, ab \rangle$$

$$\langle a(a \mid a)^{\overline{*}}, ab \rangle$$

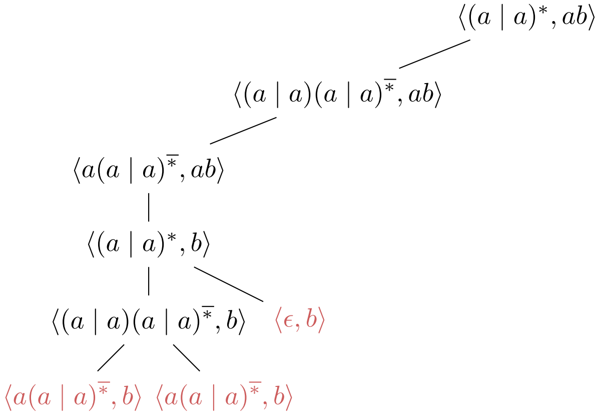
$$\langle (a \mid a)^*, b \rangle$$

$$\langle (a \mid a)(a \mid a)^{\overline{*}}, b \rangle$$

$$\langle a(a \mid a)^{\overline{*}}, b \rangle \quad \langle a(a \mid a)^{\overline{*}}, b \rangle$$

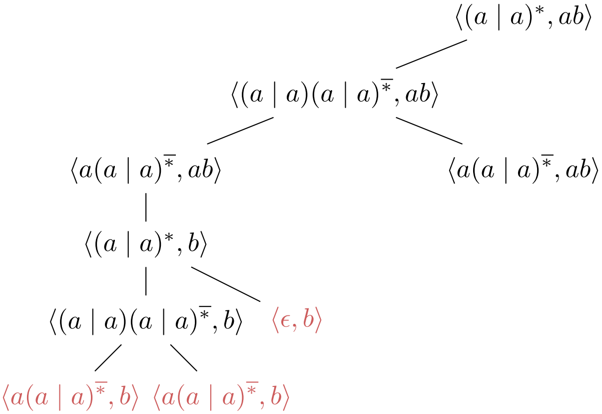
~ Semantics of regex matching

Let $R \in Regex$ and $w \in Words$. We define $[\![R]\!](w)$ as a tree.



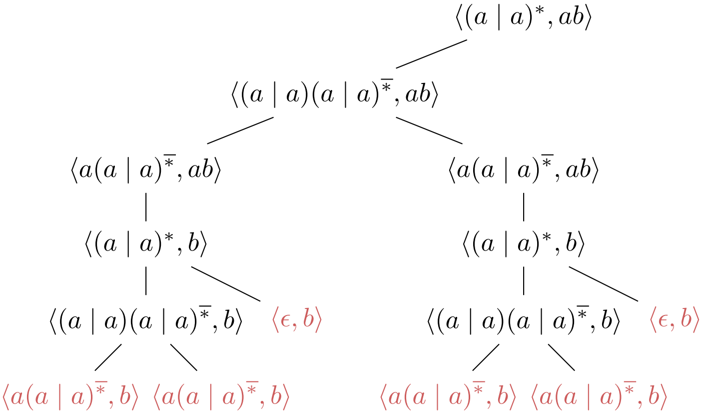
~ Semantics of regex matching

Let $R \in Regex$ and $w \in Words$. We define $[\![R]\!](w)$ as a tree.



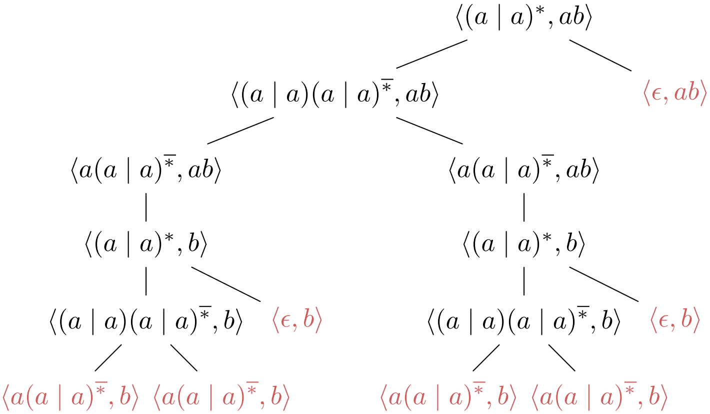
~ Semantics of regex matching

Let $R \in Regex$ and $w \in Words$. We define $[\![R]\!](w)$ as a tree.



~ Semantics of regex matching

Let $R \in Regex$ and $w \in Words$. We define $[\![R]\!](w)$ as a tree.



~ ReDoS Detection ~

~ Vulnerabilities

Definition

R has a **ReDoS vulnerability** iff the size of the trees generated by $[\![R]\!]$ grows exponentially with the length of the strings

~ Vulnerabilities

Definition

R has a **ReDoS vulnerability** iff the size of the trees generated by $[\![R]\!]$ grows exponentially with the length of the strings

Intuition: stars with nondeterminism are dangerous

~ Vulnerabilities

Definition

R has a **ReDoS vulnerability** iff the size of the trees generated by $[\![R]\!]$ grows exponentially with the length of the strings

Intuition: stars with nondeterminism are dangerous

- $(a \mid a)^*$ matches ab expanding **two** traces

~ Vulnerabilities

Definition

R has a **ReDoS vulnerability** iff the size of the trees generated by $[\![R]\!]$ grows exponentially with the length of the strings

Intuition: stars with nondeterminism are dangerous

- $(a \mid a)^*$ matches ab expanding **two** traces
- $(a \mid a)^*$ matches aab expanding **four** traces

~ Vulnerabilities

Definition

R has a **ReDoS vulnerability** iff the size of the trees generated by $[\![R]\!]$ grows exponentially with the length of the strings

Intuition: stars with nondeterminism are dangerous

- $(a \mid a)^*$ matches ab expanding **two** traces
- $(a \mid a)^*$ matches aab expanding **four** traces
- In general, $a^n b$ with 2^n traces

~ Towards ReDoS Detection

Function to capture nondeterminism:

$$\mathcal{M}_2(R) = \{w \in Words \mid \text{there are two traces to match } w \text{ in } R\}$$

~ Towards ReDoS Detection

Function to capture nondeterminism:

$$\mathcal{M}_2(R) = \{w \in Words \mid \text{there are two traces to match } w \text{ in } R\}$$

An algorithm `M2` to compute it:

$$(a \mid a)^*$$

~ Towards ReDoS Detection

Function to capture nondeterminism:

$$\mathcal{M}_2(R) = \{w \in Words \mid \text{there are two traces to match } w \text{ in } R\}$$

An algorithm `M2` to compute it:

$$(a \mid a)^* \dashrightarrow (a \mid a)(a \mid a)^* \cap \epsilon = \emptyset$$

~ Towards ReDoS Detection

Function to capture nondeterminism:

$$\mathcal{M}_2(R) = \{w \in Words \mid \text{there are two traces to match } w \text{ in } R\}$$

An algorithm `M2` to compute it:

$$(a \mid a)^* \dashrightarrow (a \mid a)(a \mid a)^* \cap \epsilon = \emptyset$$

$$\downarrow$$

$$(a \mid a)(a \mid a)^*$$

~ Towards ReDoS Detection

Function to capture nondeterminism:

$$\mathcal{M}_2(R) = \{w \in Words \mid \text{there are two traces to match } w \text{ in } R\}$$

An algorithm `M2` to compute it:

$$\begin{array}{ccc} (a \mid a)^* & \dashrightarrow & (a \mid a)(a \mid a)^* \cap \epsilon = \emptyset \\ \downarrow & & \\ (a \mid a)(a \mid a)^* & \dashrightarrow & a(a \mid a)^* \cap a(a \mid a)^* = aa^* \end{array}$$

~ Towards ReDoS Detection

Function to capture nondeterminism:

$$\mathcal{M}_2(R) = \{w \in Words \mid \text{there are two traces to match } w \text{ in } R\}$$

An algorithm `M2` to compute it:

$$\begin{array}{ccc}
(a \mid a)^* & \dashrightarrow & (a \mid a)(a \mid a)^* \cap \epsilon = \emptyset \\
\downarrow & & \\
(a \mid a)(a \mid a)^* & \dashrightarrow & a(a \mid a)^* \cap a(a \mid a)^* = aa^* \\
\downarrow & & \\
a(a \mid a)^* & &
\end{array}$$

~ Towards ReDoS Detection

Function to capture nondeterminism:

$$\mathcal{M}_2(R) = \{w \in Words \mid \text{there are two traces to match } w \text{ in } R\}$$

An algorithm `M2` to compute it:

$$\begin{array}{ccc}
(a \mid a)^* & \dashrightarrow & (a \mid a)(a \mid a)^* \cap \epsilon = \emptyset \\
\downarrow & & \\
(a \mid a)(a \mid a)^* & \dashrightarrow & a(a \mid a)^* \cap a(a \mid a)^* = aa^* \\
\downarrow & & \\
a(a \mid a)^* & \dashrightarrow & \text{Atoms do not introduce nondeterminism}
\end{array}$$

~ ReDoS detection

Structural induction on R

$$a(a|a)^*b$$

~ ReDoS detection

Structural induction on R

$$a(a|a)^*b$$

$$\swarrow$$

a ✓

~ ReDoS detection

Structural induction on R

$$a(a|a)^*b$$

$$\swarrow \qquad \searrow$$

$$a \checkmark \qquad (a|a)^*b$$

~ ReDoS detection

Structural induction on R

$$a(a|a)^*b$$

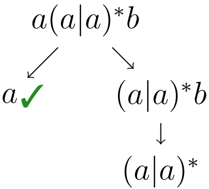
$$\swarrow \qquad \searrow$$

$$a \checkmark \qquad (a|a)^*b$$

$$\downarrow$$

$$(a|a)^*$$

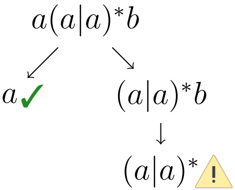
~ ReDoS detection



Structural induction on R

Run `M2` on each star of R

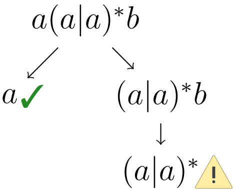
~ ReDoS detection



Structural induction on R

Run `M2` on each star of R

~ ReDoS detection



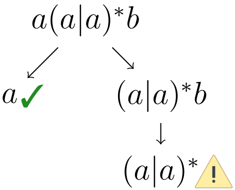
Structural induction on R

Run `M2` on each star of R

Return an overapproximation of **attack language**:

$$\mathcal{E}(R) \in Regex$$

~ ReDoS detection



Structural induction on R

Run `M2` on each star of R

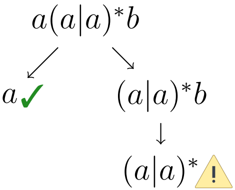
Return an overapproximation of **attack language**:

$$\mathcal{E}(R) \in Regex$$

Theorem (Soundness)

If $\mathcal{E}(R)$ is empty, then the size of matching trees grows at most polynomially with the length of input words

~ ReDoS detection



Structural induction on R

Run `M2` on each star of R

Return an overapproximation of **attack language**:

$$\mathcal{E}(R) \in Regex$$

Theorem (Soundness)

If $\mathcal{E}(R)$ is empty, then the size of matching trees grows at most polynomially with the length of input words

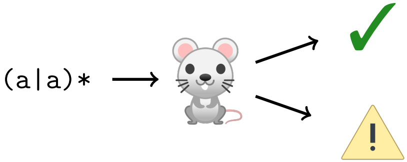
The other direction does not hold (**no completeness**)

Possible false positives, but **no false negatives**

~ Experimental Evaluation ~

~ Experimental comparison

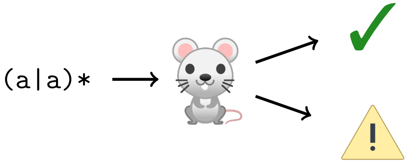
Implementation: `rat`



~ Experimental comparison

Implementation: `rat`

Compared to seven other detectors

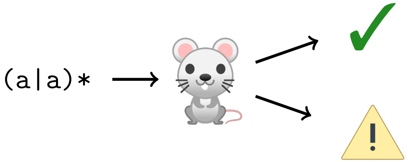


~ Experimental comparison

Implementation: `rat`

Compared to seven other detectors

Dataset of **74,670** regexes



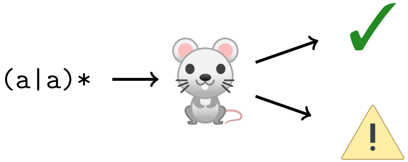
~ Experimental comparison

Implementation: `rat`

Compared to seven other detectors

Dataset of **74,670** regexes

Found **316** vulnerabilities



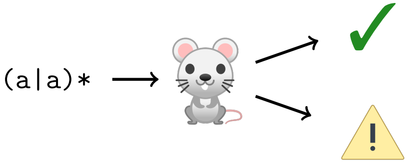
~ Experimental comparison

Implementation: `rat`

Compared to seven other detectors

Dataset of **74,670** regexes

Found **316** vulnerabilities



	OOT	SKIP	TIME	FP	FN
`rat`	178	7,390	1:57:20	49	0
`rxxr2` [1]	10	13,765	0:09:29	93	7
`rsa` [2]	789	16,177	18:48:02	193	1
`rsa-full` [2]	3,138	16,139	38:11:07	134	1
`rexploiter` [3]	328	20,202	9:12:34	28	180
`rescue` [4]	32,208	8,890	325:00:26	0	40
`safe-regex`	0	0	0:15:40	13,376	21
`regexploit`	2	421	0:03:41	56	140
`redos-detector`	2	14,749	0:52:27	14,218	6

1. Static analysis for regular expression exponential runtime via substructural logics. Rathnayake and Thielecke. 2014.
2. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. Weideman et al. 2016.
3. Static detection of dos vulnerabilities in programs that use regular expressions. Wüstholz et al. 2017.
4. ReScue: crafting regular expression DoS attacks. Shen et al. 2018.

~ Part II: Safety-Nonexploitability Analysis ~

~ Introduction ~

~ Abstract interpretation

Technique to **prove** the absence of runtime errors

~ Abstract interpretation

Technique to **prove** the absence of runtime errors

Sound but **not complete**

~ Abstract interpretation

Technique to **prove** the absence of runtime errors

Sound but **not complete**

Too many FPs $\implies$ meaningless results

~ Abstract interpretation

Technique to **prove** the absence of runtime errors

Sound but **not complete**

Too many FPs $\implies$ meaningless results

To lower FPs: more precise abstractions

~ False positives in most analyzers

The ASTRÉE Analyzer⋆

Patrick Cousot [2], Radhia Cousot [1,3], Jerôme Feret [2], Laurent Mauborgne [2], Antoine Miné [2], David Monniaux [1,2] & Xavier Rival [2]

[1] CNRS
[2] École Normale Supérieure, Paris, France (Firstname.Lastname@ens.fr)
[3] École Polytechnique, Palaiseau, France (Firstname.Lastname@polytechnique.fr)

http://www.astree.ens.fr/

Abstract. ASTRÉE is an abstract interpretation-based static program analyzer aiming at proving automatically the absence of run time errors in programs written in the C programming language. It has been applied with success to large embedded control-command safety critical real-time software generated automatically from synchronous specifications, producing a correctness proof for complex software without any false alarm in a few hours of computation.

~ False positives in most analyzers

The ASTRÉE Analyzer*

Patrick Cousot [2], Radhia Cousot [1,3], Jerôme Feret [2], Laurent Mauborgne [2], Antoine Miné [2], David Monniaux [1,2] & Xavier Rival [2]

[1] CNRS
[2] École Normale Supérieure, Paris, France (Firstname.Lastname@ens.fr)
[3] École Polytechnique, Palaiseau, France (Firstname.Lastname@polytechnique.fr)

http://www.astree.ens.fr/

Abstract. ASTRÉE is an abstract interpretation-based static program analyzer aiming at proving automatically the absence of run time errors in programs written in the C programming language. It has been applied with success to large embedded control-command safety critical real-time software generated automatically from synchronous specifications, producing a correctness proof for complex software without any false alarm in a few hours of computation.

Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer*

Matthieu Journault[1], Antoine Miné[1,2], Raphaël Monat[1], and Abdelraouf Ouadjaout[1]

[1] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
firstname.lastname@lip6.fr
[2] Institut Universitaire de France, F-75005, Paris, France

Abstract. We discuss the design of MOPSA, an ongoing effort to design a novel semantic static analyzer by abstract interpretation. MOPSA strives to achieve a high degree of modularity and extensibility by considering value abstractions for numeric, pointer, objects, arrays, etc. as well as syntax-driven iterators and control-flow abstractions uniformly as domain modules, which offer a unified signature and loose coupling, so that they can be combined and reused at will. Moreover, domains can dynamically rewrite expressions, which simplifies the design of relational abstractions, encourages a design based on layered semantics, and enables domain reuse across different analyses and different languages. We present preliminary applications of MOPSA analyzing simple programs in subsets of the C and Python programming languages, checking them for run-time errors and uncaught exceptions.

```
Checks summary: 12738 total, ✓ 12519 safe, ✗ 2 errors, ⚠ 217 warnings
  Stub condition: 66 total, ✓ 32 safe, ⚠ 34 warnings
  Invalid memory access: 6086 total, ✓ 5952 safe, ✗ 1 error, ⚠ 133 warnings
  Division by zero: 10 total, ✓ 10 safe
  Integer overflow: 6363 total, ✓ 6319 safe, ⚠ 44 warnings
  Invalid shift: 86 total, ✓ 86 safe
  Invalid pointer comparison: 1 total, ✗ 1 error
  Insufficient variadic arguments: 1 total, ✓ 1 safe
  Insufficient format arguments: 71 total, ✓ 68 safe, ⚠ 3 warnings
  Invalid type of format argument: 54 total, ✓ 51 safe, ⚠ 3 warnings
```

~ Not all bugs are equal

```c
void use_input(char* input) {
  char dest[10];
  strcpy(dest, input); // Error!
}

void main() {
  char buff[100];
  use_input(buff);
}
```

~ Not all bugs are equal

```c
void use_input(char* input) {
  char dest[10];
  strcpy(dest, input); // Error!
}


void main() {
  char buff[100];
  use_input(buff);
}
```

```c
void use_input(char* input) {
  char dest[10];
  strcpy(dest, input);
}

void main() {
  char buff[100];
  fgets(buff, sizeof(buff), stdin);
  use_input(buff);
}
```

~ Not all bugs are equal

```c
void use_input(char* input) {
  char dest[10];
  strcpy(dest, input); // Error!
}

void main() {
  char buff[100];
  use_input(buff);
}
```

```c
void use_input(char* input) {
  char dest[10];
  strcpy(dest, input);
}

void main() {
  char buff[100];
  fgets(buff, sizeof(buff), stdin);
  use_input(buff);
}
```

Security errors are more dangerous

~ Sound abstract safety-nonexploitability analysis

Lower number of alarms by **reporting only security-related ones**

~ Sound abstract safety-nonexploitability analysis

Lower number of alarms by **reporting only security-related ones**

New hyperproperty: **safety-nonexploitability**

~ Sound abstract safety-nonexploitability analysis

Lower number of alarms by **reporting only security-related ones**

New hyperproperty: **safety-nonexploitability**

Sound static analysis

~ Sound abstract safety-nonexploitability analysis

Lower number of alarms by **reporting only security-related ones**

New hyperproperty: **safety-nonexploitability**

Sound static analysis

Implementation and experiments

~ Safety-nonexploitability ~

~ Syntax

$$
\begin{aligned}
\texttt{S} &:= \texttt{x = A} && \text{(Programs)}\\
&\mid \boxed{\texttt{x = input()}}\\
&\mid \boxed{\texttt{x = rand()}}\\
&\mid \texttt{S ; S}\\
&\mid \texttt{if (B) S else S}\\
&\mid \texttt{while (B) S}\\
\texttt{A} &:= \texttt{x} && \text{(Arithmeitc Expressions)}\\
&\mid \texttt{n}\\
&\mid \texttt{A} \diamond \texttt{A} \quad (\diamond \in \{\texttt{+, -, *, /}\})\\
\texttt{B} &:= \texttt{A} < \texttt{A} && \text{(Boolean Expressions)}
\end{aligned}
$$

~ Semantics

$$
\begin{aligned}
\mathtt{x} &\in \mathbb{V} && \text{(Variables)}\\
m &\in \mathbb{M} = \mathbb{V} \to \mathbb{Z} && \text{(Memories)}\\
\langle m, i, r \rangle &\in \mathbb{S} = \mathbb{M} \times \mathbb{Z}^{\infty} \times \mathbb{Z}^{\infty} && \text{(States)}\\
[\![\mathtt{S}]\!] &\in \mathbb{D} = \mathbb{S} \to \mathbb{S} && \text{(Semantics)}
\end{aligned}
$$

~ Semantics

$$
\begin{aligned}
\mathtt{x} &\in \mathbb{V} && \text{(Variables)} \\
m &\in \mathbb{M} = \mathbb{V} \to \mathbb{Z} && \text{(Memories)} \\
\langle m, i, r \rangle &\in \mathbb{S} = \mathbb{M} \times \mathbb{Z}^{\infty} \times \mathbb{Z}^{\infty} && \text{(States)} \\
[\![\mathtt{S}]\!] &\in \mathbb{D} = \mathbb{S} \to \mathbb{S} && \text{(Semantics)}
\end{aligned}
$$

Error states are **explicitly** represented as states with $\mathtt{return} = 1$

~ Safety-nonexploitability: a formal definition

The user cannot **interfere with the correctness** of the program

~ Safety-nonexploitability: a formal definition

The user cannot **interfere with the correctness** of the program

Changing only user input does not change `return`

~ Safety-nonexploitability: a formal definition

The user cannot **interfere with the correctness** of the program

Changing only user input does not change `return`

$$\mathscr{NE} = \{[\![\mathtt{S}]\!] \mid \forall \langle\langle m_0, i_0, r_0\rangle, \langle m_1, i_1, r_1\rangle\rangle, \langle\langle m_0', i_0', r_0'\rangle, \langle m_1', i_1', r_1'\rangle\rangle \in [\![\mathtt{S}]\!] : \\ m_0 = m_0', r_0 = r_0' \implies m_1[\mathtt{return}] = m_1'[\mathtt{return}]\}$$

~ Safety-nonexploitability: a formal definition

The user cannot **interfere with the correctness** of the program

Changing only user input does not change `return`

$$\mathscr{NE} = \{[\![\mathtt{S}]\!] \mid \forall \langle\langle m_0, i_0, r_0\rangle, \langle m_1, i_1, r_1\rangle\rangle, \langle\langle m'_0, i'_0, r'_0\rangle, \langle m'_1, i'_1, r'_1\rangle\rangle \in [\![\mathtt{S}]\!] : \\ m_0 = m'_0, r_0 = r'_0 \implies m_1[\mathtt{return}] = m'_1[\mathtt{return}]\}$$

```
x = input()
1 / x
```

Safety-exploitable

```
x = rand()
1 / x
```

Safety-nonexploitable

~ Proving $\mathscr{NE}$ ~

~ Semantic user-input dependency

A variable is **tainted** iff the user can control its value

~ Semantic user-input dependency

A variable is **tainted** iff the user can control its value

$$\mathscr{T}(\mathtt{x}) = \{[\![\mathtt{S}]\!] \mid \exists\langle\langle m_0, i_0, r_0\rangle, \langle m_1, i_1, r_1\rangle\rangle, \langle\langle m'_0, i'_0, r'_0\rangle, \langle m'_1, i'_1, r'_1\rangle\rangle \in [\![\mathtt{S}]\!] : \\ m_0 = m'_0, r_0 = r'_0 \wedge m_1[\mathtt{x}] \neq m'_1[\mathtt{x}]\}$$

~ Semantic user-input dependency

A variable is **tainted** iff the user can control its value

$$\mathscr{T}(\mathtt{x}) = \{[\![\mathtt{S}]\!] \mid \exists \langle\langle m_0, i_0, r_0\rangle, \langle m_1, i_1, r_1\rangle\rangle, \langle\langle m_0', i_0', r_0'\rangle, \langle m_1', i_1', r_1'\rangle\rangle \in [\![\mathtt{S}]\!] : \\ m_0 = m_0', r_0 = r_0' \wedge m_1[\mathtt{x}] \neq m_1'[\mathtt{x}]\}$$

$$\alpha_t(\mathscr{R}) = \{\mathtt{x} \mid \forall [\![\mathtt{S}]\!] \in \mathscr{R} : [\![\mathtt{S}]\!] \in \mathscr{T}(\mathtt{x})\}$$

~ Semantic user-input dependency

A variable is **tainted** iff the user can control its value

$$\mathscr{T}(\mathtt{x}) = \{[\![\mathtt{S}]\!] \mid \exists \langle\langle m_0, i_0, r_0\rangle, \langle m_1, i_1, r_1\rangle\rangle, \langle\langle m_0', i_0', r_0'\rangle, \langle m_1', i_1', r_1'\rangle\rangle \in [\![\mathtt{S}]\!] : \\ m_0 = m_0', r_0 = r_0' \wedge m_1[\mathtt{x}] \neq m_1'[\mathtt{x}]\}$$

$$\alpha_t(\mathscr{R}) = \{\mathtt{x} \mid \forall [\![\mathtt{S}]\!] \in \mathscr{R} : [\![\mathtt{S}]\!] \in \mathscr{T}(\mathtt{x})\} \quad \mathtt{x} \text{ is tainted in } \mathtt{S} \overset{\triangle}{\iff} \mathtt{x} \in \alpha_t(\{[\![\mathtt{S}]\!]\})$$

~ Safety-nonexploitability and taint

$$[\![\texttt{S}]\!] \in \mathscr{N\!E} \iff \texttt{return} \text{ is not tainted in } \texttt{S}$$

~ Safety-nonexploitability and taint

$$[\![\texttt{S}]\!] \in \mathscr{NE} \iff \texttt{return} \text{ is not tainted in } \texttt{S}$$

Idea: overapproximate the semantics, and pair it with **sound** taint analysis

~ Analysis ~

~ A semantic taint analysis

Regular value domains

- find RTEs

Taint domain

- label RTEs as exploitable

~ A semantic taint analysis

Regular value domains

- find RTEs

Taint domain

- label RTEs as exploitable

Side effect

- enhanced taint precision

~ A semantic taint analysis

```
x = input()
// tainted = {x}
```

Regular value domains

- find RTEs

Taint domain

- label RTEs as exploitable

Side effect

- enhanced taint precision

~ A semantic taint analysis

```
x = input()
// tainted = {x}
if (x == 1) {
  y = 1
} else {
  y = 0
}
```

Regular value domains

- find RTEs

Taint domain

- label RTEs as exploitable

Side effect

- enhanced taint precision

~ A semantic taint analysis

Regular value domains

- find RTEs

Taint domain

- label RTEs as exploitable

Side effect

- enhanced taint precision

```
x = input()
// tainted = {x}
if (x == 1) {
  y = 1
} else {
  y = 0
}
// tainted = {x,y}
```

~ A semantic taint analysis

Regular value domains

- find RTEs

Taint domain

- label RTEs as exploitable

Side effect

- enhanced taint precision

```
x = input()
// tainted = {x}
if (x == 1) {
  y = 1
} else {
  y = 0
}
// tainted = {x,y}
1 / y
```

~ A semantic taint analysis

Regular value domains

- find RTEs

Taint domain

- label RTEs as exploitable

Side effect

- enhanced taint precision

```
x = input()
// tainted = {x}
if (x == 1) {
  y = 1
} else {
  y = 0
}
// tainted = {x,y}
1 / y    ⚠
```

~ A semantic taint analysis

Regular value domains

- find RTEs

Taint domain

- label RTEs as exploitable

Side effect

- enhanced taint precision

```
x = input()
// tainted = {x}
if (x == 1) {
  y = 1
} else {
  y = 0
}
// tainted = {x,y}
1 / y    ⚠
x = y
```

~ A semantic taint analysis

Regular value domains

- find RTEs

Taint domain

- label RTEs as exploitable

Side effect

- enhanced taint precision

```
x = input()
// tainted = {x}
if (x == 1) {
  y = 1
} else {
  y = 0
}
// tainted = {x,y}
1 / y    ⚠
x = y
z = x - y
```

~ A semantic taint analysis

Regular value domains

- find RTEs

Taint domain

- label RTEs as exploitable

Side effect

- enhanced taint precision

```
x = input()
// tainted = {x}
if (x == 1) {
  y = 1
} else {
  y = 0
}
// tainted = {x,y}
1 / y    ⚠
x = y
z = x - y
// tainted = {x,y}
```

~ A semantic taint analysis

Regular value domains

- find RTEs

Taint domain

- label RTEs as exploitable

Side effect

- enhanced taint precision

```
x = input()
// tainted = {x}
if (x == 1) {
  y = 1
} else {
  y = 0
}
// tainted = {x,y}
1 / y    ⚠
x = y
z = x - y
// tainted = {x,y}
1 / z
```

~ A semantic taint analysis

Regular value domains

- find RTEs

Taint domain

- label RTEs as exploitable

Side effect

- enhanced taint precision

```
x = input()
// tainted = {x}
if (x == 1) {
  y = 1
} else {
  y = 0
}
// tainted = {x,y}
1 / y    ⚠
x = y
z = x - y
// tainted = {x,y}
1 / z    NE ✓
```

~ Experimental Evaluation ~

~ Implementation

Implementation: `Mopsa-Nexp`

Finds common C safety (exploitable) errors

Based on `Mopsa`

~ Experiments

77 **real-world** programs from Coreutils

- Up to ~4000 LOCs

Compared **precision** and **performance** of `Mopsa-Nexp` with `Mopsa`

~ Experiments

77 **real-world** programs from Coreutils

- Up to ~4000 LOCs

Compared **precision** and **performance** of `Mopsa-Nexp` with `Mopsa`

Analyzer	**Alarms**	**Time**
`Mopsa`	$4,715$	1:17:06
`Mopsa-Nexp`	$1,217$	1:28:42

~ Experiments

77 **real-world** programs from Coreutils

- Up to ~4000 LOCs

Compared **precision** and **performance** of `Mopsa-Nexp` with `Mopsa`

Analyzer	Alarms	Time
`Mopsa`	$4,715$	1:17:06
`Mopsa-Nexp`	$1,217$	1:28:42

- We prove **~74%** of the alarms nonexploitable
- Performance overhead: **<16%**

~ Related Work ~

~ Related Work

Noninterference [5]

- $\mathscr{NE}$ can be seen as noninterference of `return`
- We can prove noninterference with our analysis
- Techniques for noninterference are not sufficient for $\mathscr{NE}$

~ Related Work

Noninterference [5]

- $\mathscr{NE}$ can be seen as noninterference of `return`
- We can prove noninterference with our analysis
- Techniques for noninterference are not sufficient for $\mathscr{NE}$

Taint analysis [6]

- We rely on a semantic definition
- Combination with values
- Sinks in $\mathscr{NE}$ are RTEs

~ Related Work

Noninterference [5]

- $\mathscr{NE}$ can be seen as noninterference of `return`
- We can prove noninterference with our analysis
- Techniques for noninterference are not sufficient for $\mathscr{NE}$

Taint analysis [6]

- We rely on a semantic definition
- Combination with values
- Sinks in $\mathscr{NE}$ are RTEs

Robust reachability [7]

- Different handling of `rand`: $\exists$ vs $\forall$

~ Conclusions ~

~ ReDoS: contributions

Novel tree semantics

ReDoS formalization

Sound static analysis

Implementation and experiments on **real-world data**

- The analysis is fast and precise
- The **only sound detector in practice**

~ ReDoS: contributions

Novel tree semantics

ReDoS formalization

Sound static analysis

Implementation and experiments on **real-world data**

- The analysis is fast and precise
- The **only sound detector in practice**

Future work

- Polynomial ReDoS analysis
- Support for regular expression extensions
- Integration within a program analysis

~ Safety-nonexploitability: contributions

Novel property: safety-nonexploitability

Equivalent characterization with semantic taint

Sound semantic taint analysis

Implementation and experiments on **real-world data**

- Performance overhead <16%
- **Filtered >70% of the alarms**

~ Safety-nonexploitability: contributions

Novel property: safety-nonexploitability

Equivalent characterization with semantic taint

Sound semantic taint analysis

Implementation and experiments on **real-world data**

- Performance overhead <16%
- **Filtered >70% of the alarms**

Future work

- Extend nonexploitability to other properties
- Field-sensitive C taint analysis
- ReDoS-nonexploitability analysis

~ Conclusions

Security matters

Security poses non-trivial challenges

Formal reasoning is the only way to **ensure** security

~ References I

Asiri Rathnayake and Hayo Thielecke.
Static analysis for regular expression exponential runtime via substructural logics.
CoRR, abs/1405.7058, 2014.

Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce W. Watson.
Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA.
In *International Conference on Implementation and Application of Automata, CIAA*, volume 9705 of *Lecture Notes in Computer Science*, pages 322–334. Springer, 2016.

Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig.
Static detection of dos vulnerabilities in programs that use regular expressions.
In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 10206 of *Lecture Notes in Computer Science*, pages 3–20, 2017.

Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu.
ReScue: crafting regular expression DoS attacks.
In *International Conference on Automated Software Engineering, ASE*, pages 225–235. ACM, 2018.

Joseph A. Goguen and José Meseguer.
Security policies and security models.
In *Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.

Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon.
Static analysis of android apps: a systematic literature review.
Inf. Softw. Technol., 88:67–95, 2017.

~ References II

Guillaume Girol, Benjamin Farinier, and Sébastien Bardin.
Not all bugs are created equal, but robust reachability can tell the difference.
In *Computer Aided Verification, CAV*, volume 12759, pages 669–693. Springer, 2021.