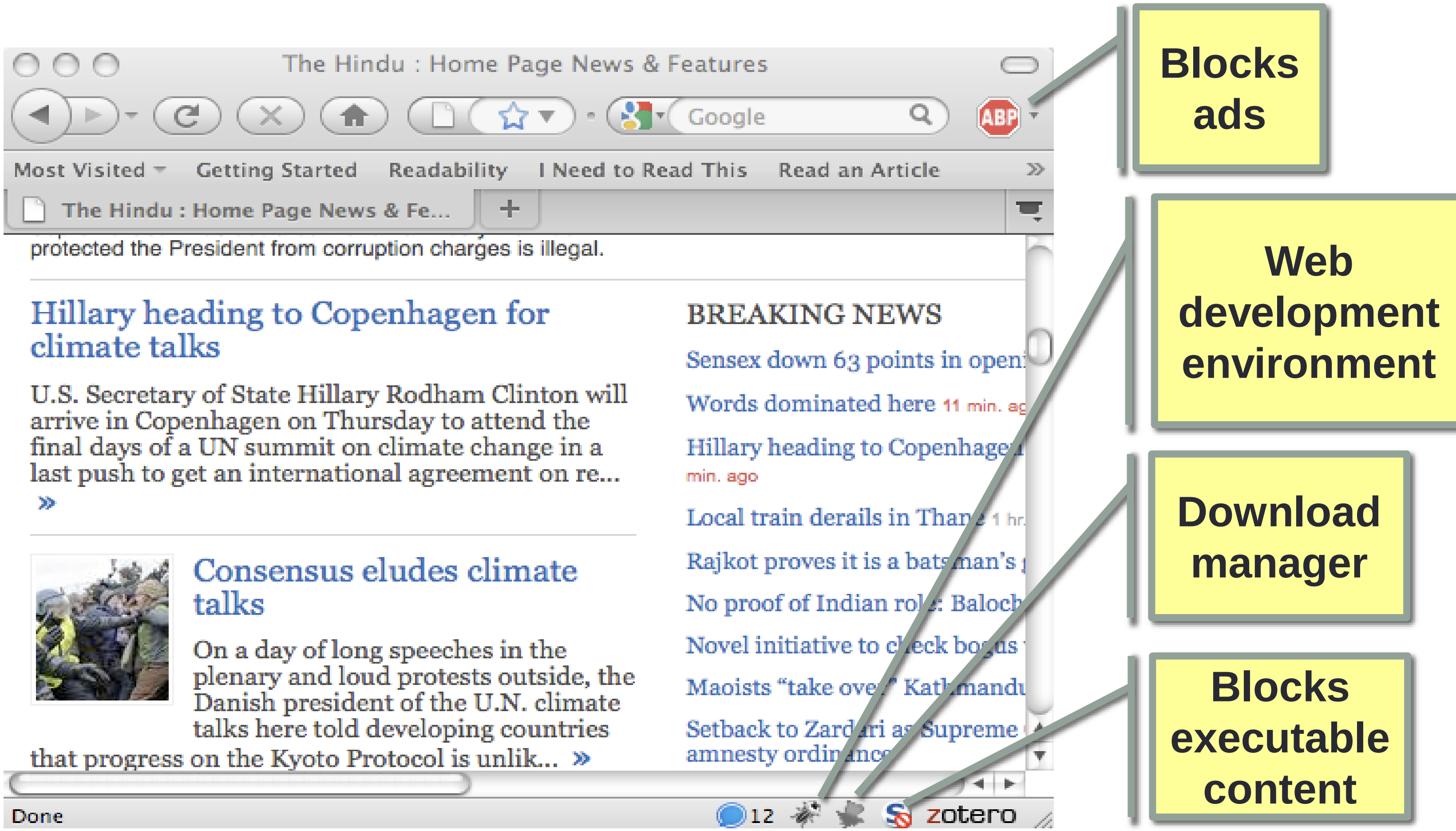
AUTOMATIC ATTACK GENERATION FOR FIREFOX EXTENSIONS

Parasara Sridhar Duggirala
Sruthi Bandhakavi
Stanley Bak
Madhusudhan Parthasarathy

Around 150 Million extensions are in use



Google chrome extensions

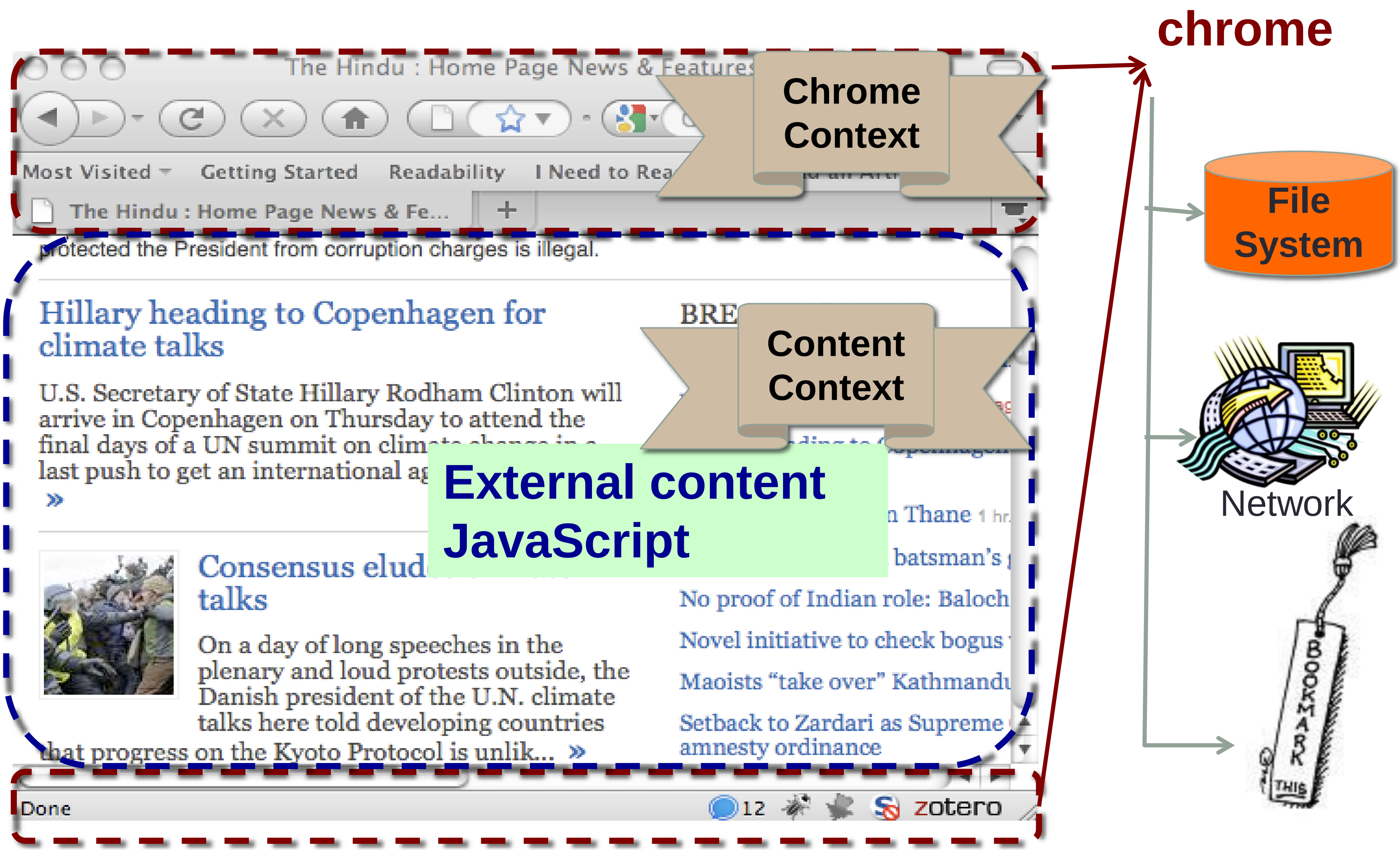
Safari Extensions

The source for Safari 5 extensions

Outline

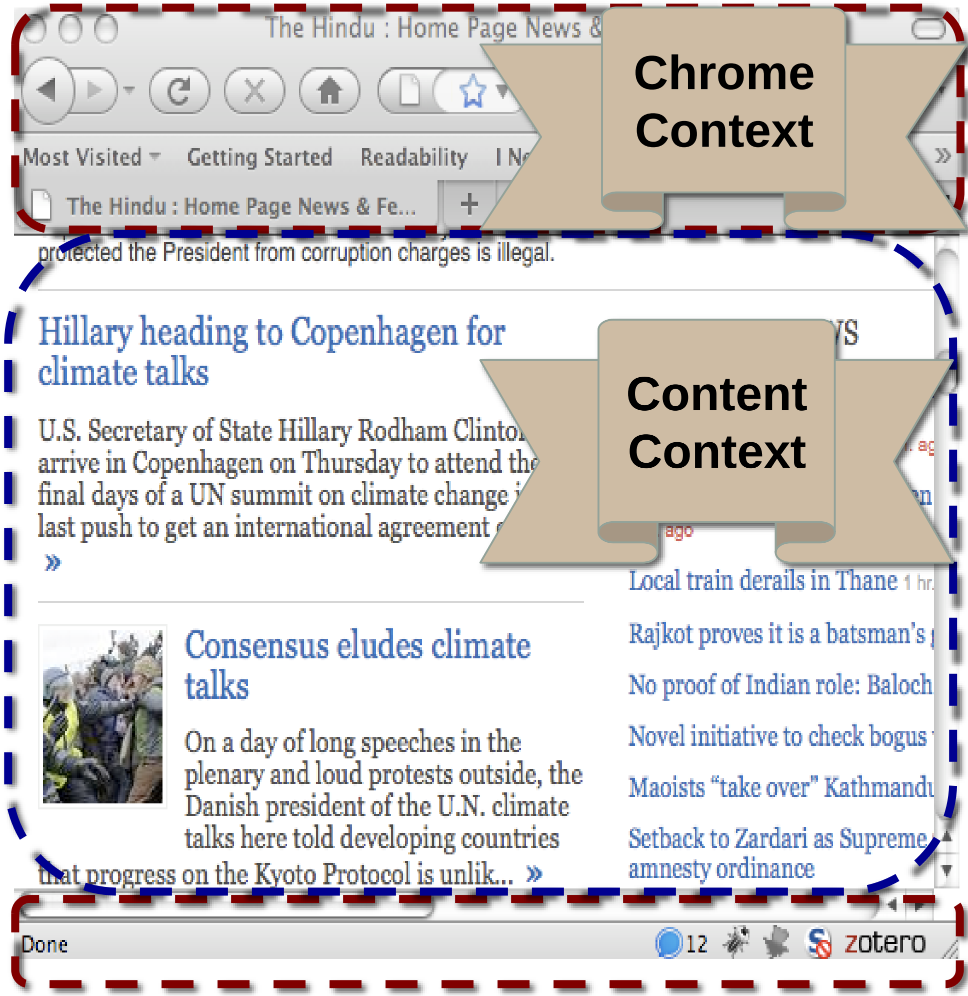
- Firefox Extensions : Background
- Attacks on extensions
- Sanitization Procedures
- String Constraint Solvers
- Feed Sidebar 3.2
- Conclusions

Working of Extensions



Attacks on Extensions

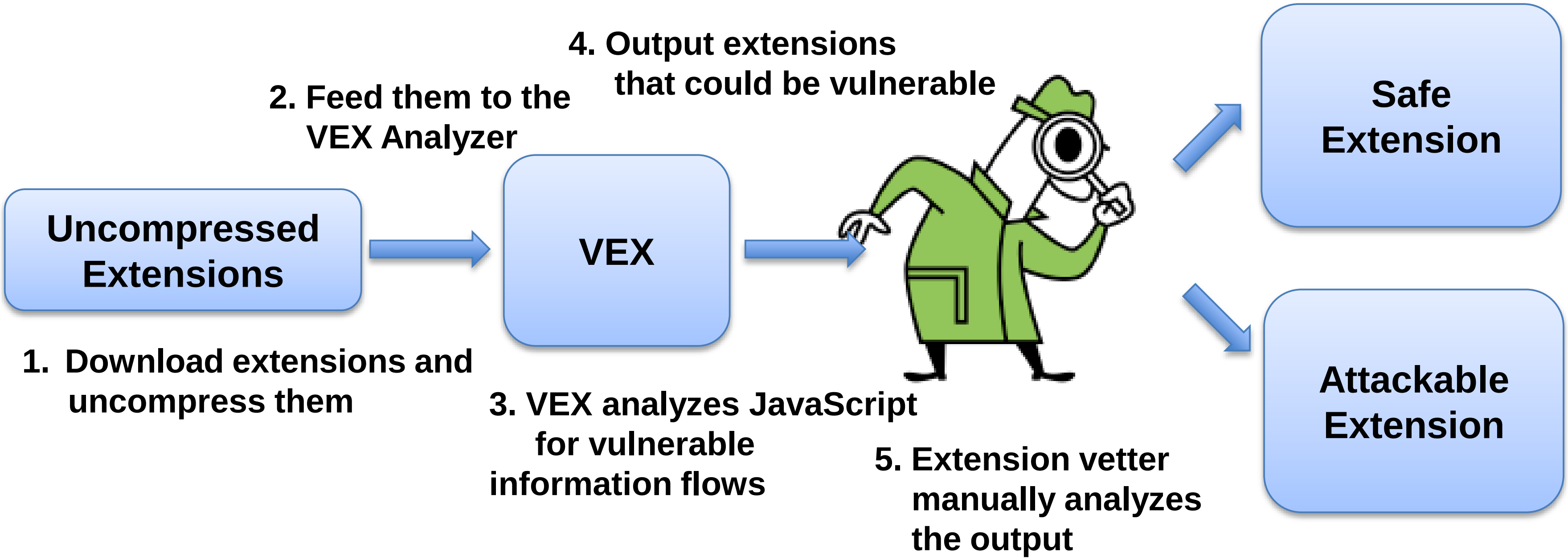
- Document Object Model
- *InnerHTML* Object
- Extension creates new *InnerHTML* object in chrome context
- Input is subjected to sanitization
- Analyze paths from the *document* to *InnerHTML*
- Attack could occur if JavaScript is executed in Chrome context
- **VEX** – a tool for vetting extensions uncovered bugs in Wikipedia toolbar, fizzle, etc…



Challenges in analyzing Extensions

- Developed by third party developers
- JavaScript – powerful language for web development
- Different purposes of extensions
- VEX – analyzes the information flow from document to InnterHTML paths

VEX – Vetting Browser Extensions For Security Vulnerabilities (USENIX Security 2010)



Detecting Vulnerable Extensions Automatically

Desired Properties

- Eliminate the manual analysis of output by Vetter
- Automatically detect whether the extension is vulnerable
- Generation of counterexamples

Tasks required

- Analyze the information flow
- Analyze the operations performed on input

Sanitization Procedures

- Procedure that sanitizes the input
- Typically consists of string manipulation techniques

Input : I; Output : O;
I' = Operations(I);
O = Sanitized(I');

- Checking sanitization procedures
  1) Fix an attack pattern P (i.e. O satisfies the pattern P)
  2) Generate string constraints
  3) Solve the string constraints using any string constraint solver

Sanitization Procedures

- Feed Sidebar 3.2

```
…
itemObject.description = itemObject.
description.replace(/<script[ˆ>]*>[\s\S]+<\/script>/gim, "");
...
```

Sanitization Procedures

- YouTube Cinema 4.8

```
…
theDescription = theDescription.replace ("\r", " ", "g");
theDescription = theDescription.replace ("\n", " ", "g");
while(theDescription.indexOf("  ") != -1)
  theDescription = theDescription.replace("  ", " ");
…
var hasHttp = (theString.indexOf ("http://") != -1);
for (var i = 0; i < theString.length; i++)
{
if (hasHttp){
if (theString.substr (i, 7) == "http://"){
j = theString.indexOf (" ", i);
if (j == -1)  j = theString.length;
j -= i;
theAddress = theString.substr (i, j);
res += ("<a class = \"fiveone\" target = \"_blank\" href = \"" + theAddress + "\">" + theAddress + "</a>");
i += (j - 1);
continue;
}
res += ("&#" + theString.charCodeAt (i) + ";");
```

Sanitization Procedures

- Sanitizes the input for a pattern
- Can be distributed throughout the extension
- Third party developers have their own procedures
- A set of procedures provided by Firefox API
- Collect all the operations on the input

Generating String Constraints

- Operations performed by Sanitization Procedures
- Commonly used string manipulation functions
  1) *indexOf*
  2) *concat*
  3) *replace*
  4) *subString*
- Next Step : To model the operations as string constraints

String Constraints

- A set of equations over strings

String : s_2, s_3;
String : $s_1 = \text{concat}(s_2, s_3)$;
ASSERT(s_1 != TEST);

- Counterexamples

s_1 = TEST, s_2 = T, s_3 = EST
s_1 = TEST, s_2 = TE, s_3 = ST
s_1 = TEST, s_2 = TES, s_3 = T

String Constraints

- String constraints involve equations over
  - Concatenation
  - Containment in Regular/context free grammar
  - Length
  - Boolean operations

Constraint := ¬ Formula
| Formula V Formula
| Formula
Formula := Var ∊ RegExp
| Var ∊ CfgExp
| Var = Var
| Var = StringConst
| Var = Var · Var
| *len*(var) Rel *len*(var)
| *len*(var) Rel *Number*
Rel := ≤ | ≥ | < | >

String Constraint Solvers

- Two classes
  - Use symbolic techniques and automata theory
  - Use BitVector constraint solvers

- Solvers that use symbolic techniques

  ~ Uses automata theoretic techniques

  + Analyze strings of unbounded length
  + Constraints in CFG and regular expressions
  + Flexible length constraints
  - No efficient tools yet

Solvers Using Bit Vector Constraints

- Hampi and Kudzu
- Hampi – for attack generation for PHP codes
- Kudzu – symbolic analysis of AJAX applications
- Can analyze strings of only bounded length

Key idea:

1. Bound length of strings for high expressiveness, efficiency
2. Typical applications require short solutions

Solvers Using Bit Vector Constraints

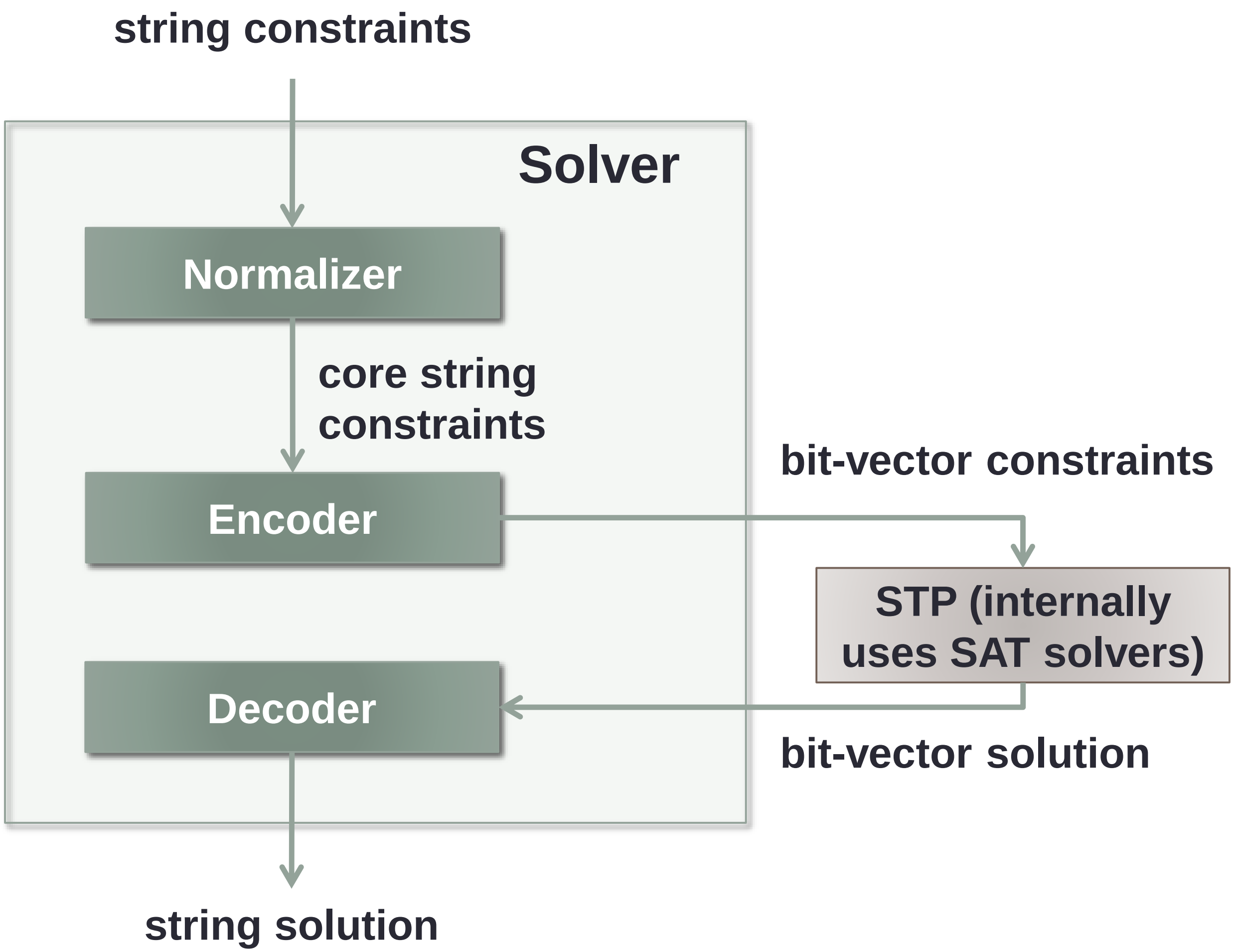
more expressive

context-free	$L_1 \cap \ldots \cap L_N$	Undecidable
regular	$R_1 \cap \ldots \cap R_N$	PSPACE-complete
bounded regular	$r_1 \cap \ldots \cap r_N$	NP-complete

more tractable

bound(any language) ➔ bounded regular

Internals of String Constraint Solver



Internals of String Constraint Solvers

var v:4;
cfg *E* := "()" | *E E* | "(" *E* ")";
var *q* := concat("(", v ,")");
q in E;
assert *q* contains "()()";

String Constraint

cfg *E* := "(" *E* ")" | *E E* | "()";

bound(*E*, 6) ➔ ([()() + (())]) + ()[()() + (())] + [()() + (())]()

Parsing and Normalizing

bit-vector constraints

Bit-vector Solver STP

B = 404041404141

Decoder

Maps bits back to alphabet Σ

B = (()())
v = ()()

Decoding

Drawbacks

- Fixed size input attack string
- Cannot handle length constraints
- Predetermined number of replaces/substrings

Back to Firefox extensions

- Looked at string constraint solvers
- Translating sanitization procedures is not trivial

```
while(theDescription.indexOf("  ") != -1)
  theDescription = theDescription.replace("  ", " ");
```

- Recursion and loops cannot be handled effectively
- Fixed size attack input?
- Scalability issues

However, there is hope

- Feed Sidebar 3.2

```
var_0xINPUT \in CapturedBrack(/.+/,0);
var_0xINPUT := P1 . P2;
P2 \in CapturedBrack(/<script[^>]*>[\s\S]+<\/script>/,0);
P1 \notin CapturedBrack(/<script[^>]*>[\s\S]+<\/script>/,0);
P3 == "";
output := P1 . P3;
test := output == "&lt;script&gt;a&lt;\/script&gt;" ;
ASSERT(test);
```

- Attack pattern : <script>a<\/script>
- Input String : <script>a<script><script>@<script>

Complete Plan

- Build a database of attacks
- Model the sanitization routines using string constraints
- Run the constraint solver for possible attacks

Drawbacks

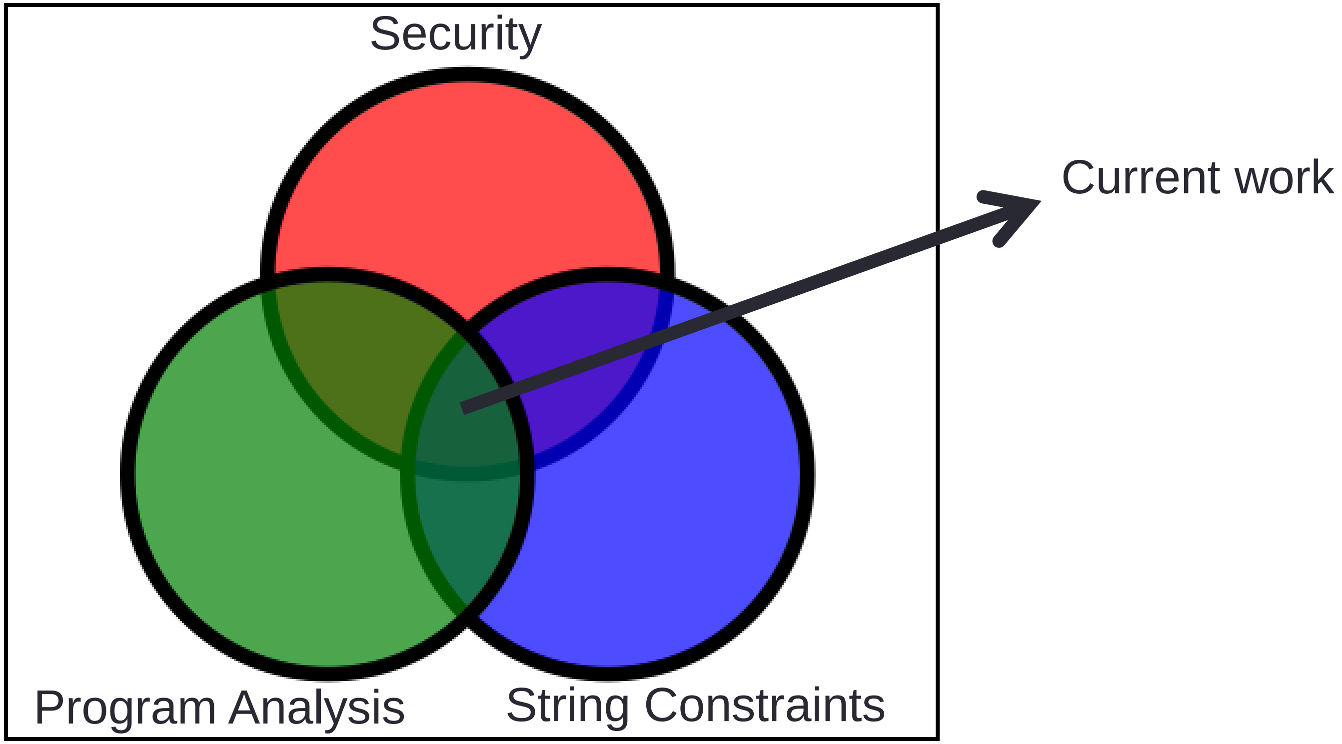
- Fixed size input
- Loops and recursive procedures

Future Work

- Explore symbolic string constraint solvers
- Improve **VEX** for collecting sanitization routines

Conclusions

- Verification of third party web applications
- Provide a counterexample for the security loophole(if any)



THANK YOU