鉴释 xcalibyte

A TOOL FOR VERIFYING BUSINESS LOGIC BEYOND COMMON VULNERABILITIES

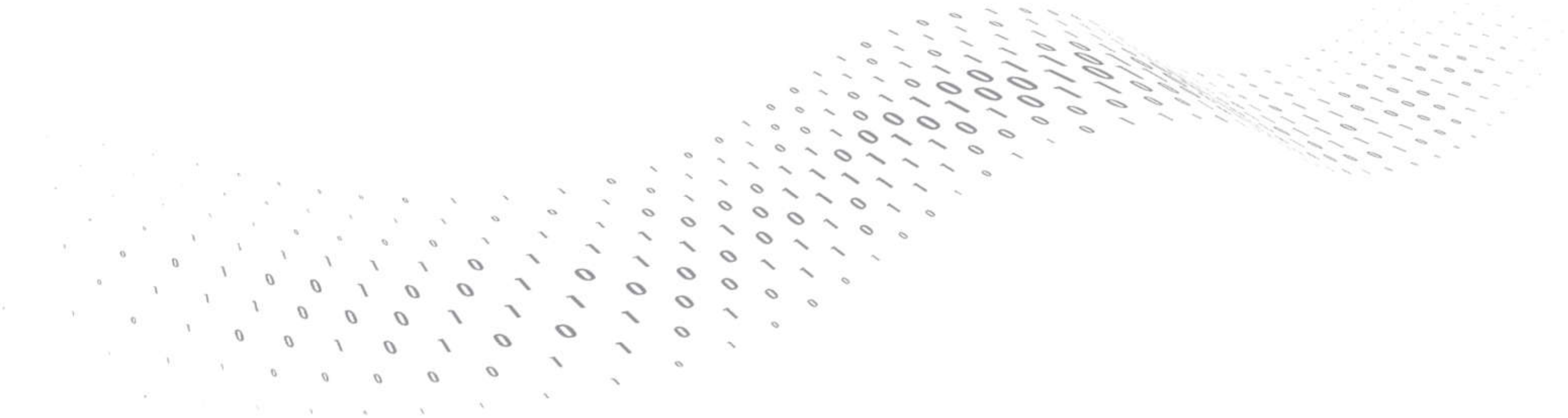
WHO ARE WE?

SHIN-MING LIU 刘新铭
鉴释首席架构师

LI LONG 李隆
鉴释首席科学家

- Compiler Scientist
- Director China Intel IOT Research Lab
- Director HP Compiler Technology Lab
- 10+ Patents granted in program analysis and compiler optimization

- PhD in CS, USTC, Software Security Laboratory
- 8+ Years HP NonStop compiler backend & SDK engineer



Xcalibyte's mission is to improve the quality of software by creating easy-to-use tools that help developers build & deploy reliable & secure code

WHY ARE WE HERE?

✓ IT Technology is at a turning point with:

- **Domain Specific Hardware** e.g. AI which will be pervasive
- Two decades of a **software boom** which needs to be sustained
- Ubiquitous **distributed computing** in a connected world

✓ Software Challenges include:

- Bugs occurring at greater frequency
- 82% of security issues are from applications
- 1 in 1000 lines of code having security Issues
- 1 in 1400 lines of code having high severity security issues

Xcalibyte, is here to help!

WHERE DO SECURITY ISSUES COME FROM?

- ✓ Vulnerabilities incubated in applications that are exposed
- ✓ Violations of underlying business logic

WHAT IS BUSINESS LOGIC?

✓ Private/sensitive data protection
  - Password
  - Personal information

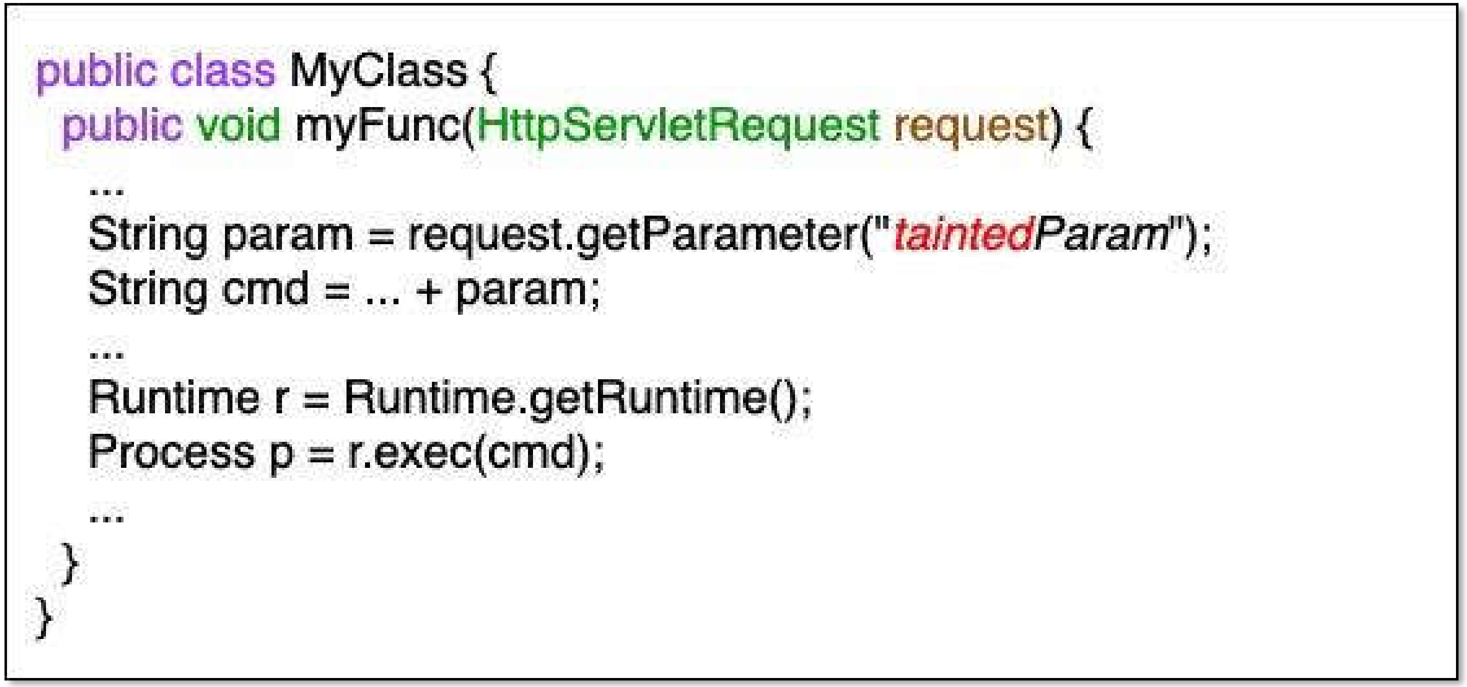
✓ Untrusted data sanitization
  - Injection

✓ Specific processing flow of data validation
  - Audit accounts for manipulation

EXAMPLE: INJECTION

Tainted data passed to Runtime.exec may cause issues

A

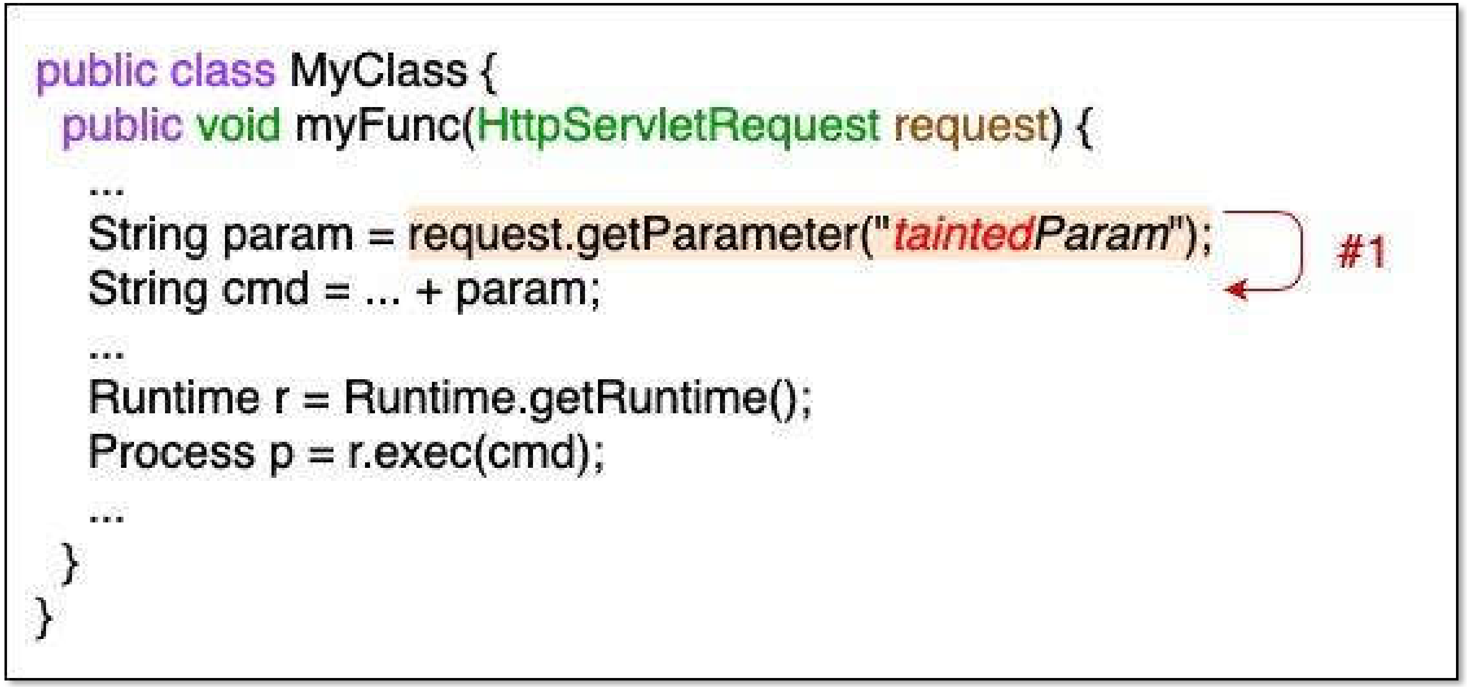


```
public class MyClass {
  public void myFunc(HttpServletRequest request) {
    ...
    String param = request.getParameter("taintedParam");
    String cmd = ... + param;
    ...
    Runtime r = Runtime.getRuntime();
    Process p = r.exec(cmd);
    ...
  }
}
```

EXAMPLE: INJECTION

Tainted data passed to Runtime.exec may cause issues

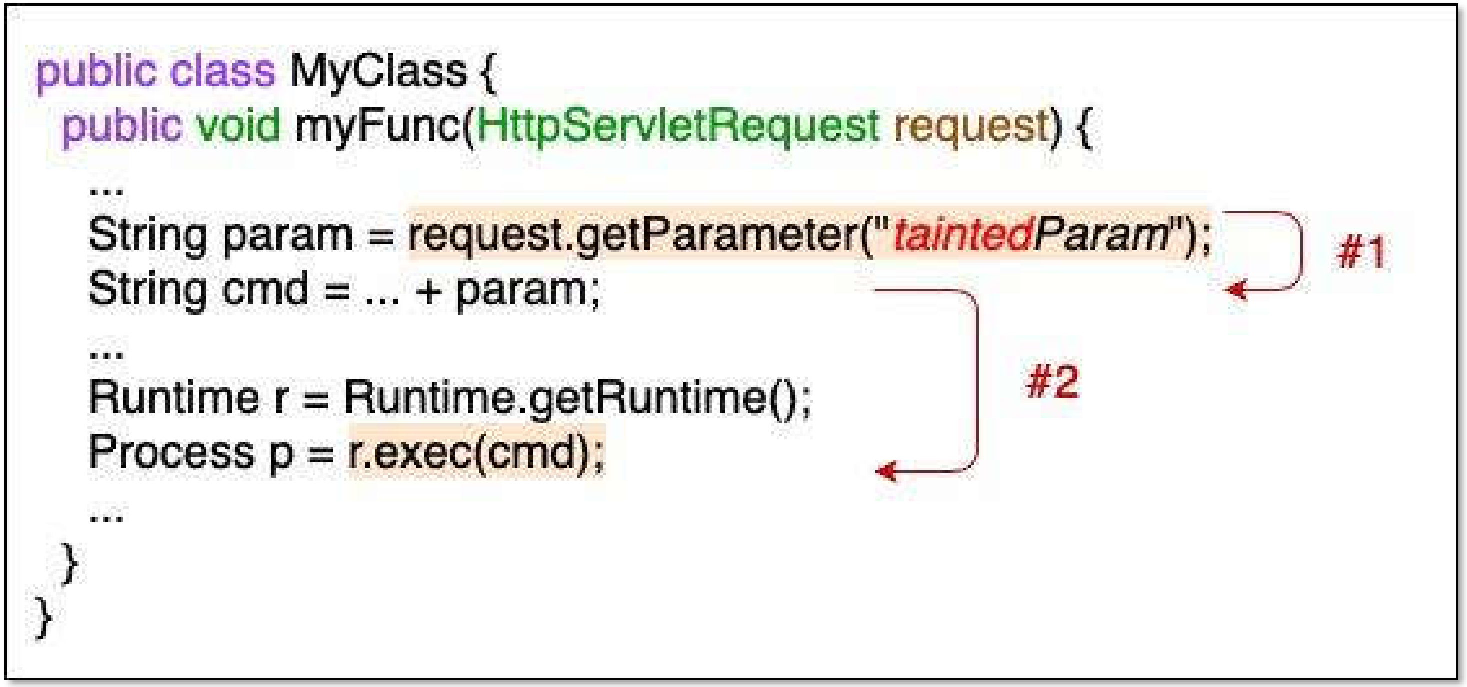
B



```
public class MyClass {
  public void myFunc(HttpServletRequest request) {
    ...
    String param = request.getParameter("taintedParam");
    String cmd = ... + param;
    ...
    Runtime r = Runtime.getRuntime();
    Process p = r.exec(cmd);
    ...
  }
}
```

EXAMPLE: INJECTION

Tainted data passed to Runtime.exec may cause issues



```
public class MyClass {
  public void myFunc(HttpServletRequest request) {
    ...
    String param = request.getParameter("taintedParam");
    String cmd = ... + param;
    ...
    Runtime r = Runtime.getRuntime();
    Process p = r.exec(cmd);
    ...
  }
}
```

WHY IS VERIFYING BUSINESS LOGIC IMPORTANT?

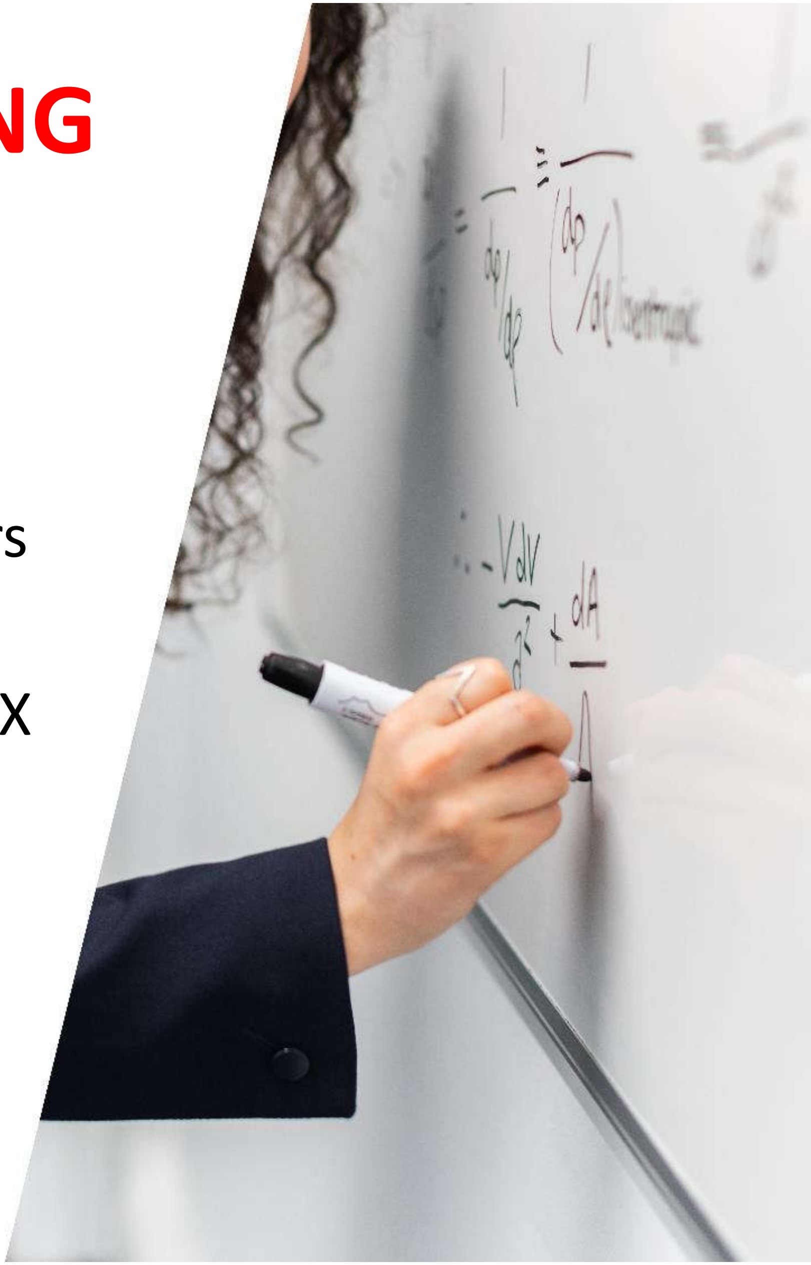
- ✓ In practice, many vulnerabilites result from violations of the underlying business logic
- ✓ Each company has specific business logic so universal checkers/ verifiers don’t always work

The ability to customize for and verify business logic is needed!

CANDIDATE: THEOREM PROVING

- Requires strict mathematical methods capable of verification tasks
- Formalizing business logic into mathematical representation is non-trivial and hard for others to understand
- ‘Proving’ efforts are very time-consuming (~10X lines of Coq proof code vs. source code)
- Highly-qualified experts are required

Expensive and unaffordable for most companies!



CANDIDATE: EXISTING SAST TOOLS

✓ Focused on revealing common vulnerabilities

- Null Pointer Dereference (NPD), Use After Free (UAF), Use Uninitialized Variable (UIV)…

✓ Compiler based (type or data flow) techniques

- Effective to discover vulnerabilities related to program syntax and feature
- But not effective to discover vulnerabilities due to program semantics and business logic

✓ Few allow customization and only on pattern match rules only

- Therefore, limited support

Company Confidential: Please do not distribute.

xcalscan PROGRAM ANALYSIS TECHNIQUES

✓ Analyze based on SSA IRs

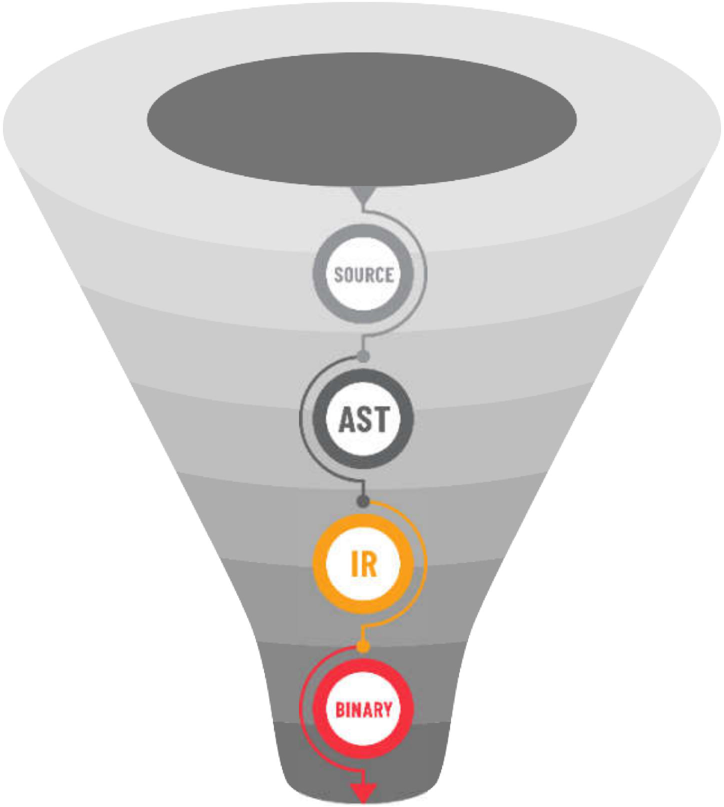
- Context sensitive
- Flow sensitive
- Cross file
- Cross language

✓ On demand analysis

- Less time
- Less memory

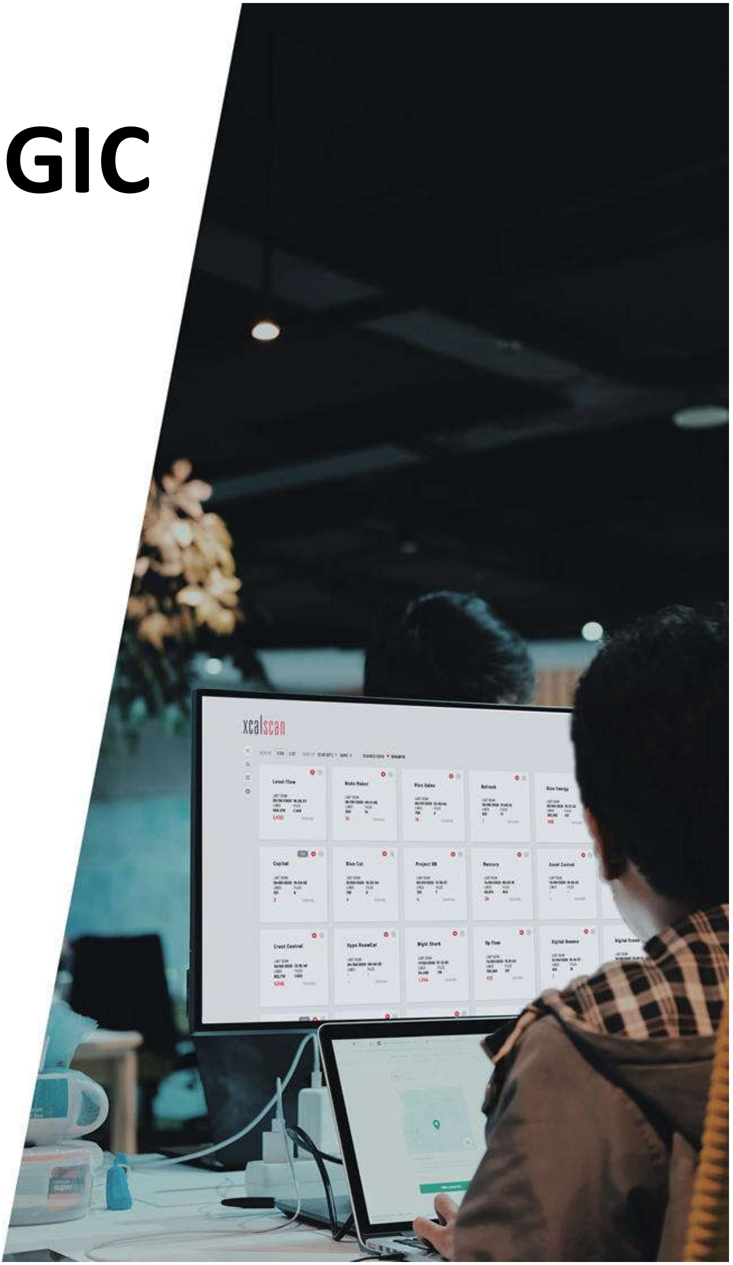
✓ Symbolic evaluation

- User customizable rules



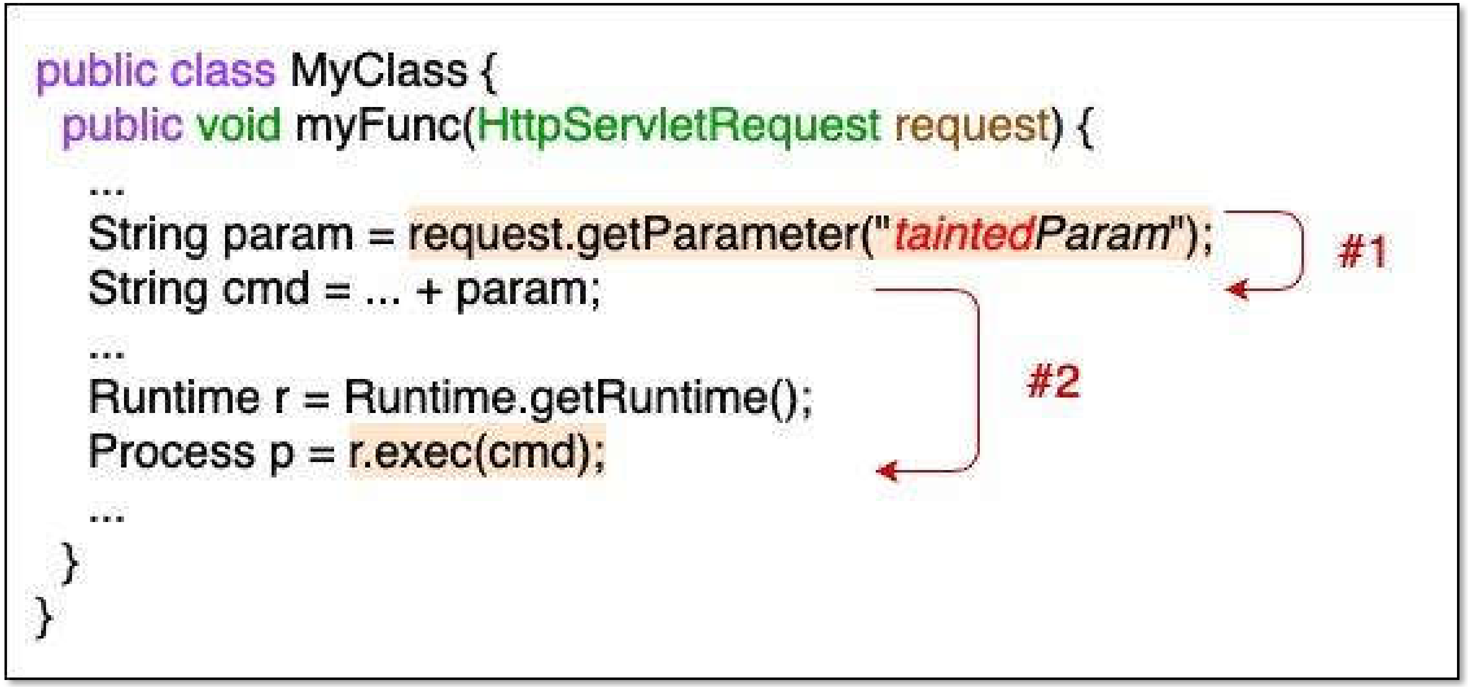
xcalscan VERIFYING BUSINESS LOGIC

- ✓ A symbolic evaluation framework
  - APIs to model side effects & to analyze user-defined rules
- ✓ No modification needed in customers’ source code
- ✓ Supports analysis without complete source code due to 3rd party API
- ✓ Customers can define their own rules via the same programming language they use during development



EXAMPLE: INJECTION - DEFECT

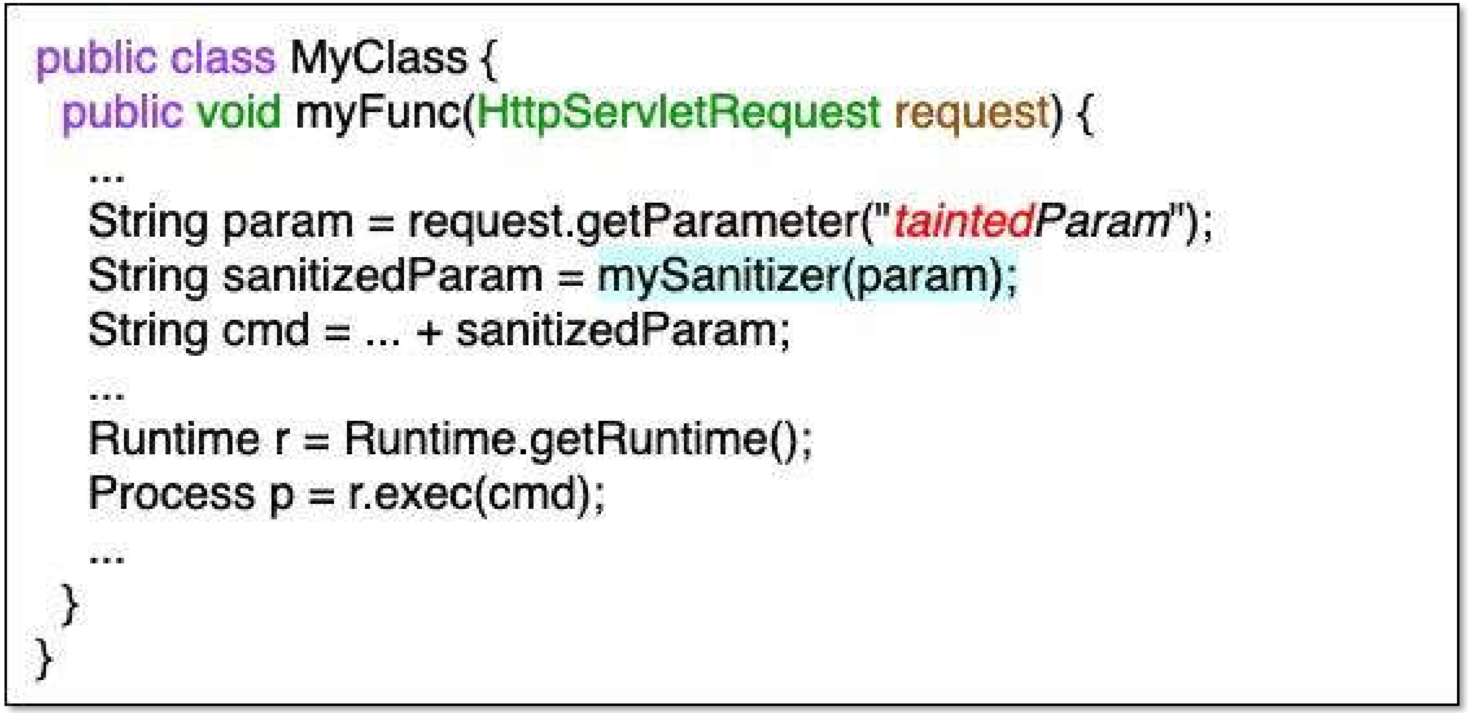
Tainted data pass to Runtime.exec may cause issues



```
public class MyClass {
  public void myFunc(HttpServletRequest request) {
    ...
    String param = request.getParameter("taintedParam");
    String cmd = ... + param;
    ...
    Runtime r = Runtime.getRuntime();
    Process p = r.exec(cmd);
    ...
  }
}
```

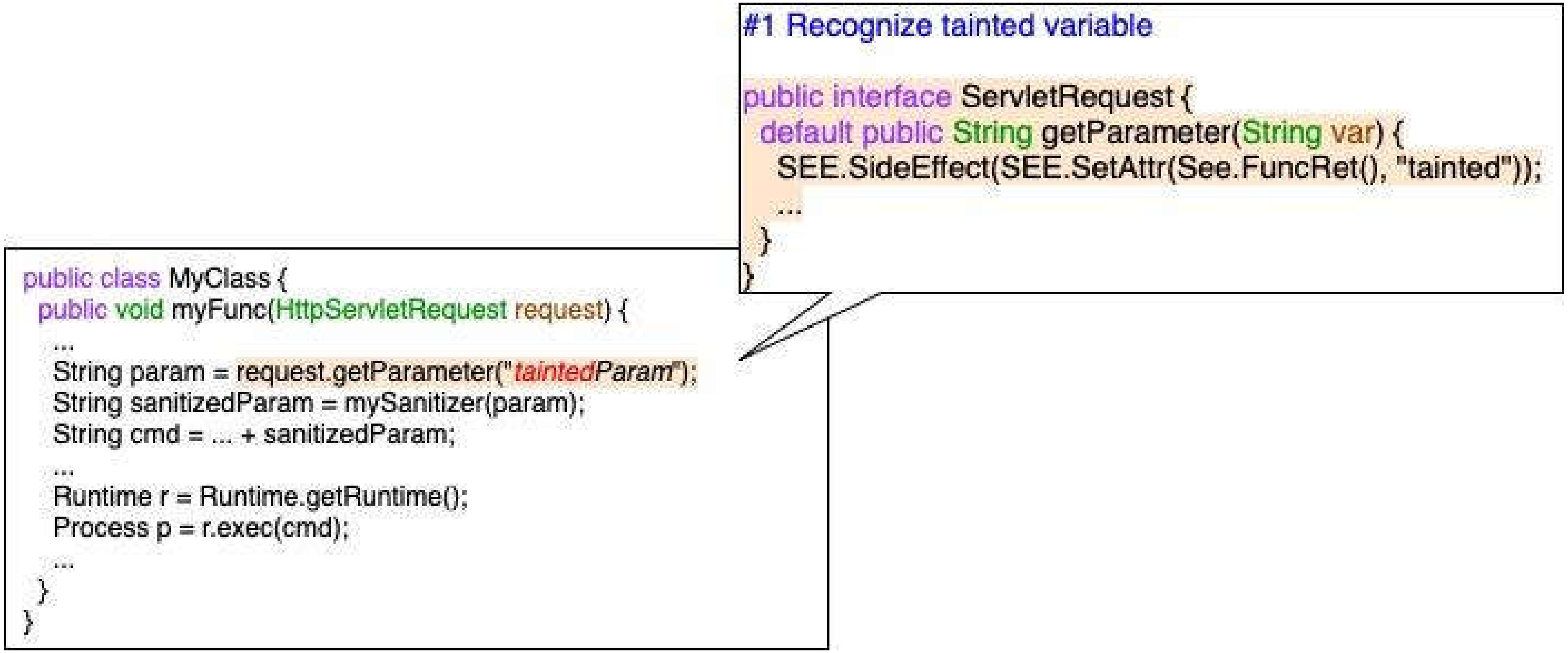
EXAMPLE: INJECTION - REMEDIATION

Sanitize tainted parameter before execution



```
public class MyClass {
 public void myFunc(HttpServletRequest request) {
   ...
   String param = request.getParameter("taintedParam");
   String sanitizedParam = mySanitizer(param);
   String cmd = ... + sanitizedParam;
   ...
   Runtime r = Runtime.getRuntime();
   Process p = r.exec(cmd);
   ...
 }
}
```

INJECTION – BUILD THE RULE, STEP 1

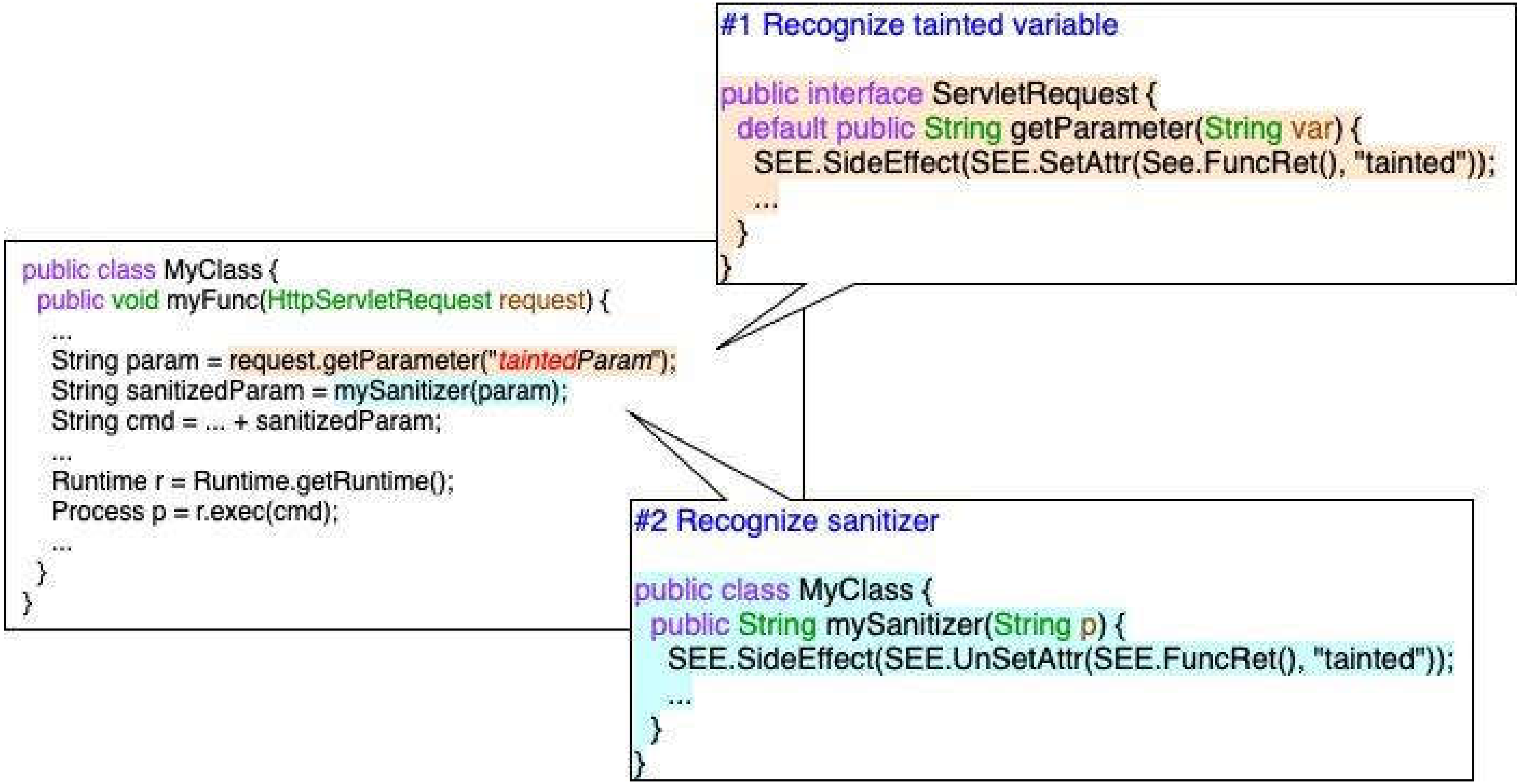


#1 Recognize tainted variable

```
public interface ServletRequest {
  default public String getParameter(String var) {
    SEE.SideEffect(SEE.SetAttr(See.FuncRet(), "tainted"));
    ...
  }
}
```

```
public class MyClass {
  public void myFunc(HttpServletRequest request) {
    ...
    String param = request.getParameter("taintedParam");
    String sanitizedParam = mySanitizer(param);
    String cmd = ... + sanitizedParam;
    ...
    Runtime r = Runtime.getRuntime();
    Process p = r.exec(cmd);
    ...
  }
}
```

INJECTION – BUILD THE RULE, STEP 2



```
public class MyClass {
  public void myFunc(HttpServletRequest request) {
    ...
    String param = request.getParameter("taintedParam");
    String sanitizedParam = mySanitizer(param);
    String cmd = ... + sanitizedParam;
    ...
    Runtime r = Runtime.getRuntime();
    Process p = r.exec(cmd);
    ...
  }
}
```

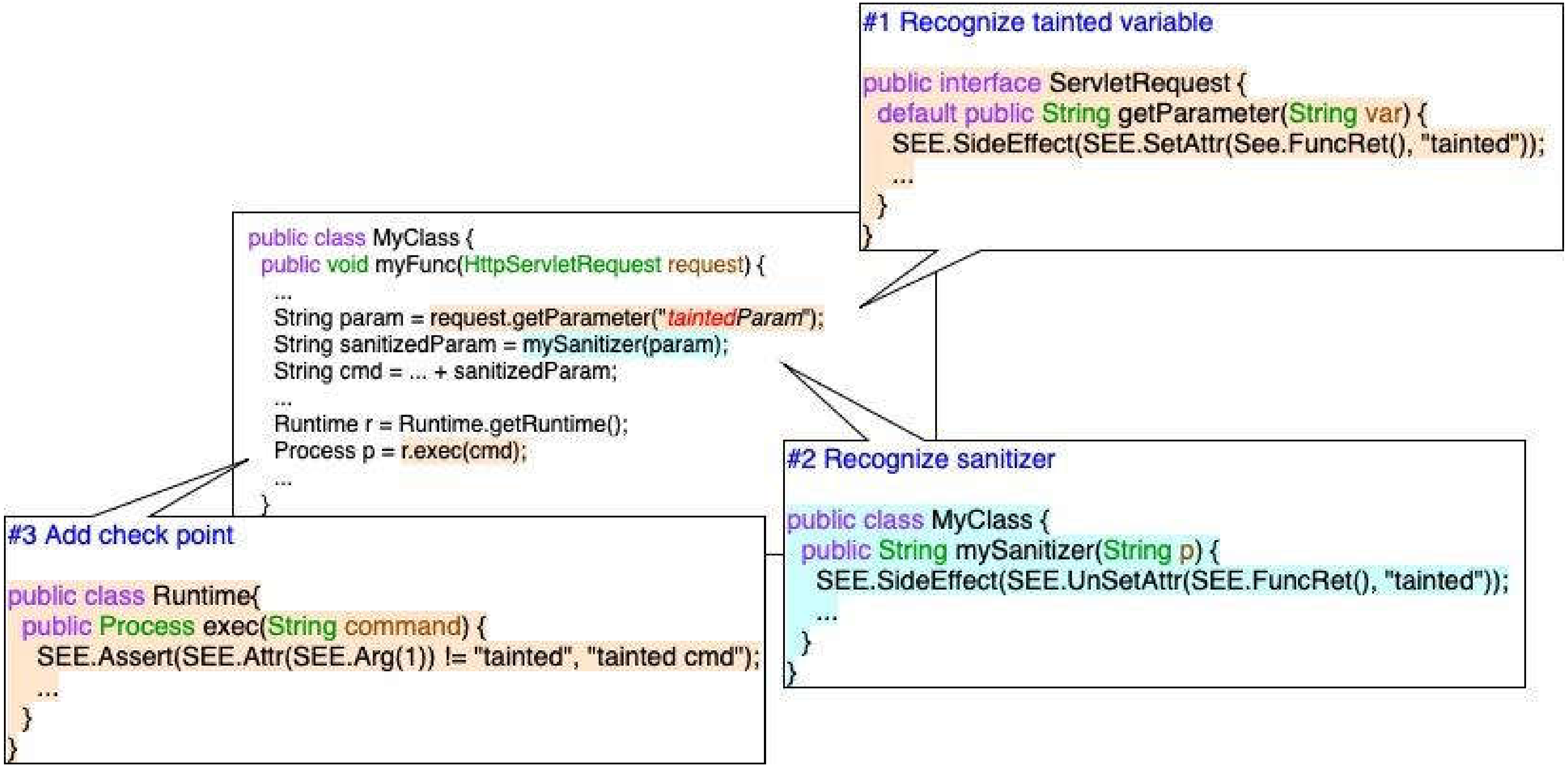
#1 Recognize tainted variable

```
public interface ServletRequest {
  default public String getParameter(String var) {
    SEE.SideEffect(SEE.SetAttr(See.FuncRet(), "tainted"));
    ...
  }
}
```

#2 Recognize sanitizer

```
public class MyClass {
  public String mySanitizer(String p) {
    SEE.SideEffect(SEE.UnSetAttr(SEE.FuncRet(), "tainted"));
    ...
  }
}
```

INJECTION – BUILD THE RULE, STEP 3



```
public class MyClass {
  public void myFunc(HttpServletRequest request) {
    ...
    String param = request.getParameter("taintedParam");
    String sanitizedParam = mySanitizer(param);
    String cmd = ... + sanitizedParam;
    ...
    Runtime r = Runtime.getRuntime();
    Process p = r.exec(cmd);
    ...
  }
```

#1 Recognize tainted variable

```
public interface ServletRequest {
  default public String getParameter(String var) {
    SEE.SideEffect(SEE.SetAttr(See.FuncRet(), "tainted"));
    ...
  }
}
```

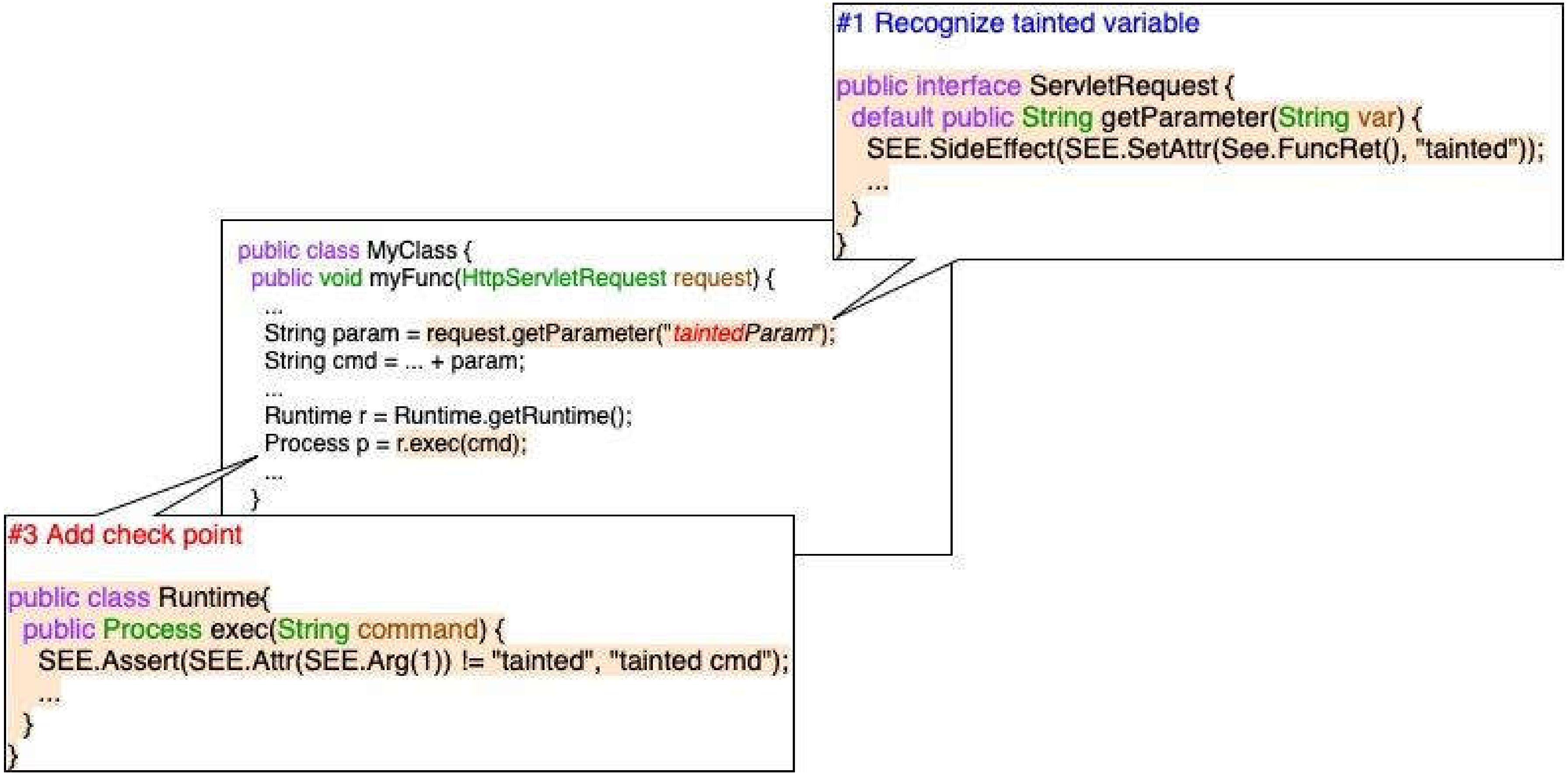
#2 Recognize sanitizer

```
public class MyClass {
  public String mySanitizer(String p) {
    SEE.SideEffect(SEE.UnSetAttr(SEE.FuncRet(), "tainted"));
    ...
  }
}
```

#3 Add check point

```
public class Runtime{
  public Process exec(String command) {
    SEE.Assert(SEE.Attr(SEE.Arg(1)) != "tainted", "tainted cmd");
    ...
  }
}
```

INJECTION – IDENTIFY THE VIOLATION



#1 Recognize tainted variable

```
public interface ServletRequest {
  default public String getParameter(String var) {
    SEE.SideEffect(SEE.SetAttr(See.FuncRet(), "tainted"));
    ...
  }
}
```

```
public class MyClass {
  public void myFunc(HttpServletRequest request) {
    ...
    String param = request.getParameter("taintedParam");
    String cmd = ... + param;
    ...
    Runtime r = Runtime.getRuntime();
    Process p = r.exec(cmd);
    ...
  }
```

#3 Add check point

```
public class Runtime{
  public Process exec(String command) {
    SEE.Assert(SEE.Attr(SEE.Arg(1)) != "tainted", "tainted cmd");
    ...
  }
}
```

xcalscan EASY CUSTOMIZATION

- A symbolic evaluation framework
  - Boolean expression: a > b …
  - Programming concepts: types, super class …
  - Busines logic: call sequence, call depth limitation …
- We define semantic descriptor for standard runtime libraries
- User functions can be bound to rules available as well
- Selectable compliance standard checks
  - CERT, CWE, OWASP …
- Customizable locations to perform checking to stay focused
  - xxCoin account overflow/underflow check

xcalscan COMMON VULNERABILITIES

✓ Analyze based on SSA IRs

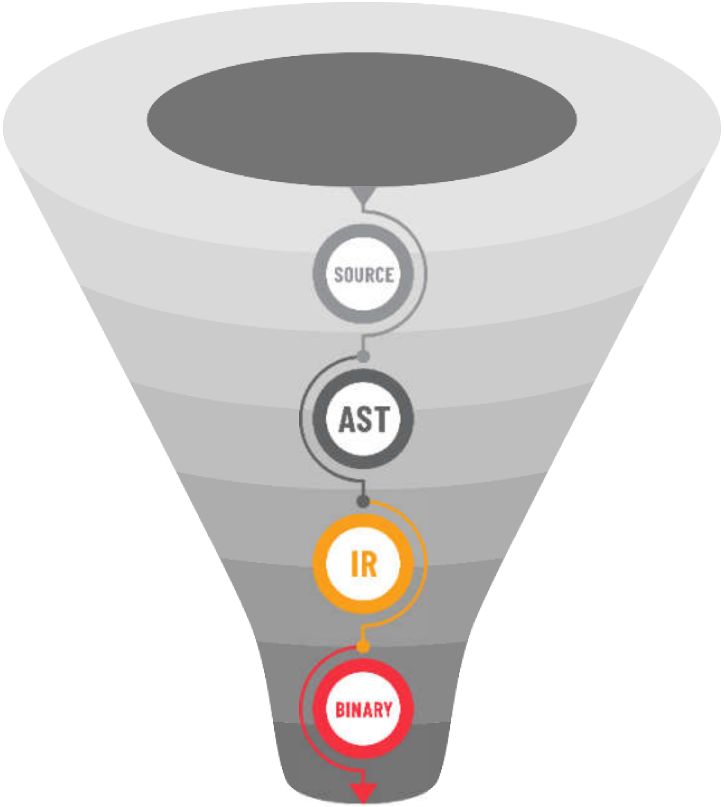
- Context sensitive
- Flow sensitive
- Cross file
- Cross language

✓ On demand analysis

- Less time
- Less memory

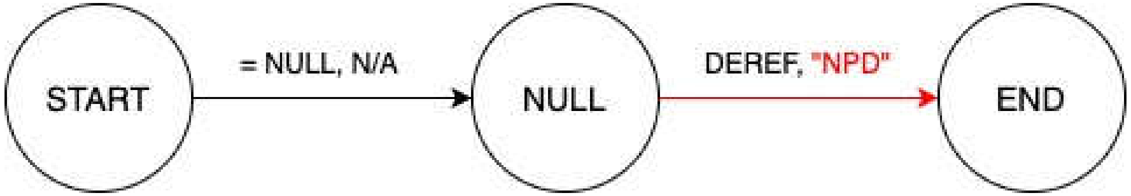
✓ Symbolic evaluation

- User customizable rules



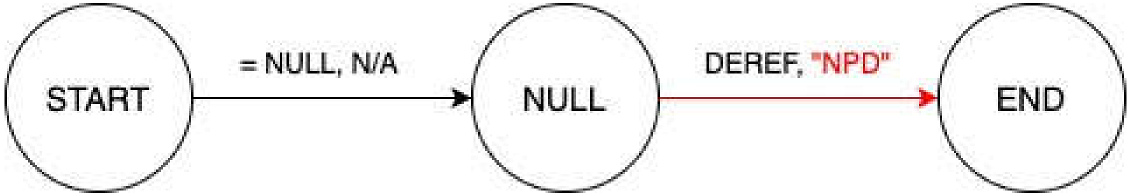
EXAMPLE: NULL POINTER DEREFERENCE FSM ABSTRACTION

A straightforward thought about the checker



EXAMPLE: NULL POINTER DEREFERENCE FSM ABSTRACTION

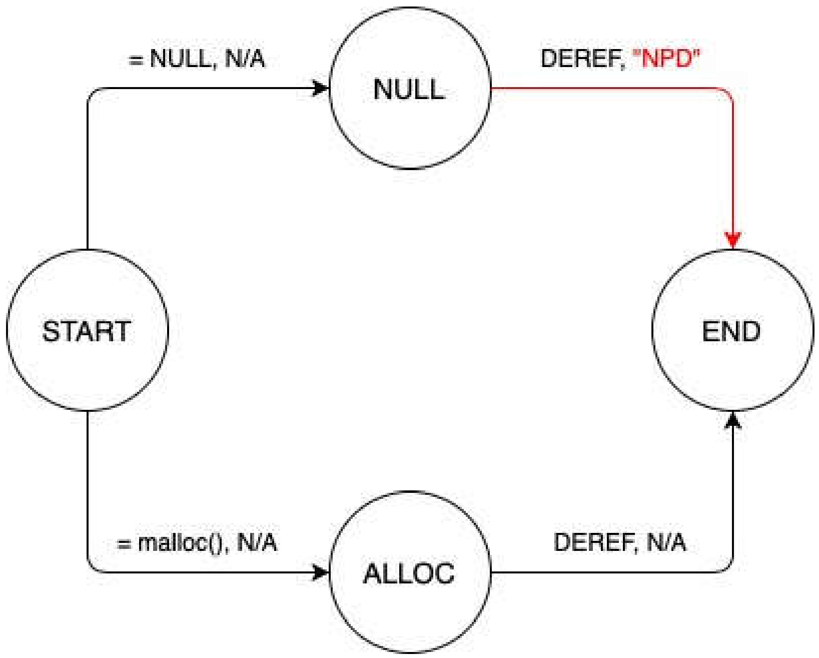
A straightforward thought about the checker



Think a little bit deeper: what’s the ‘good’ behavior?

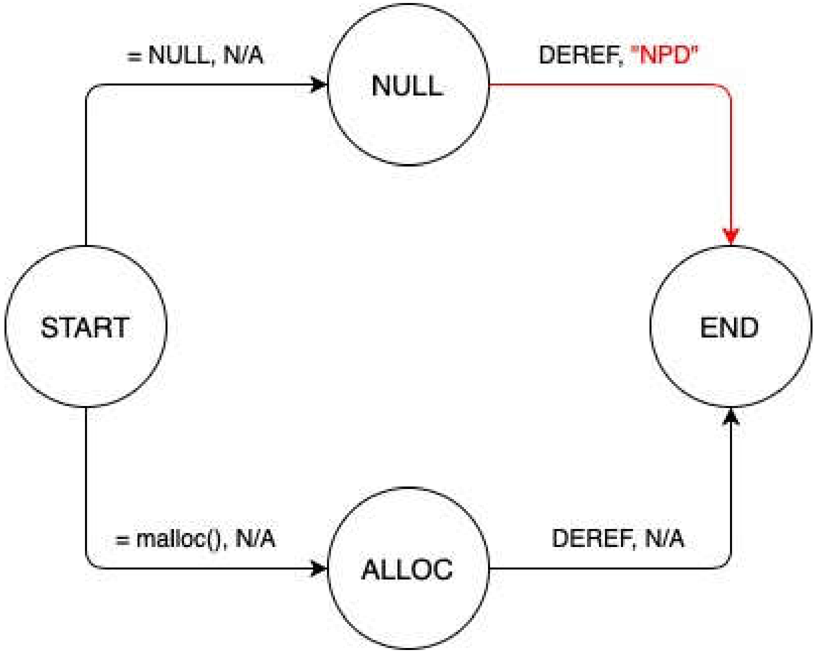
EXAMPLE: NULL POINTER DEREFERENCE FSM ABSTRACTION

‘good’ behavior: malloced pointer should be fine to dereference



EXAMPLE: NULL POINTER DEREFERENCE FSM ABSTRACTION

It is not accurate: after malloc, the pointer may be free & malloc may fail

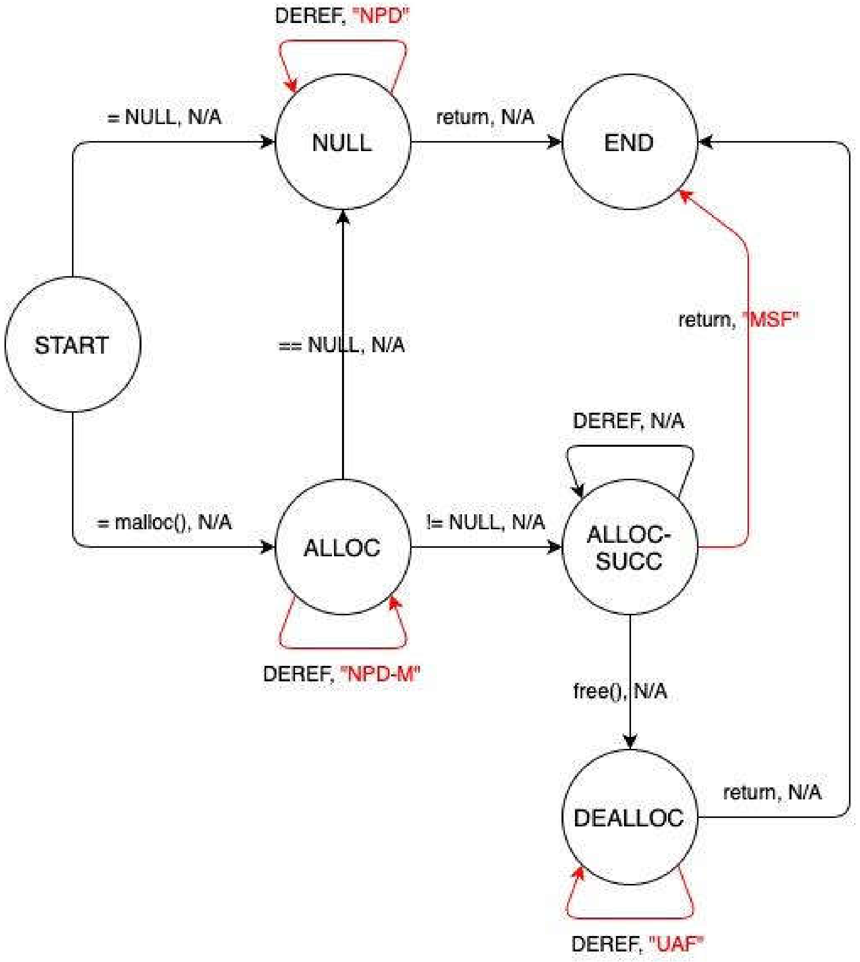


EXAMPLE: NULL POINTER DEREFERENCE FSM ABSTRACTION

Use After Freed

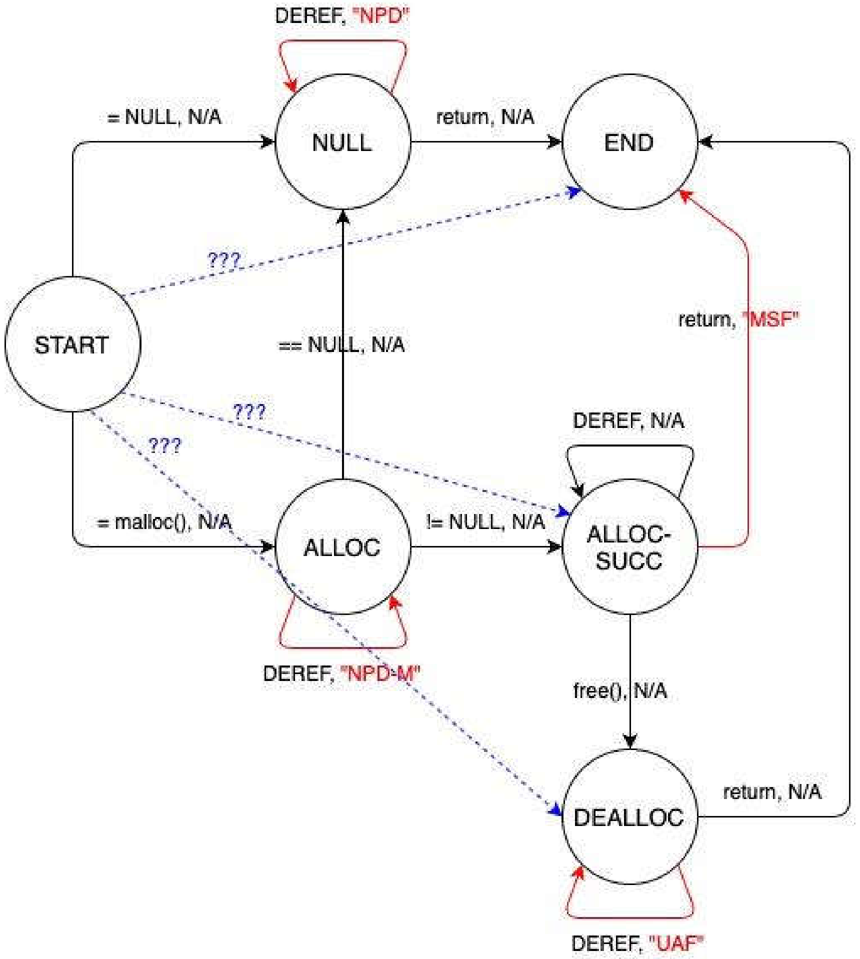
Null Pointer Dereference-Maybe

MiSsing Free



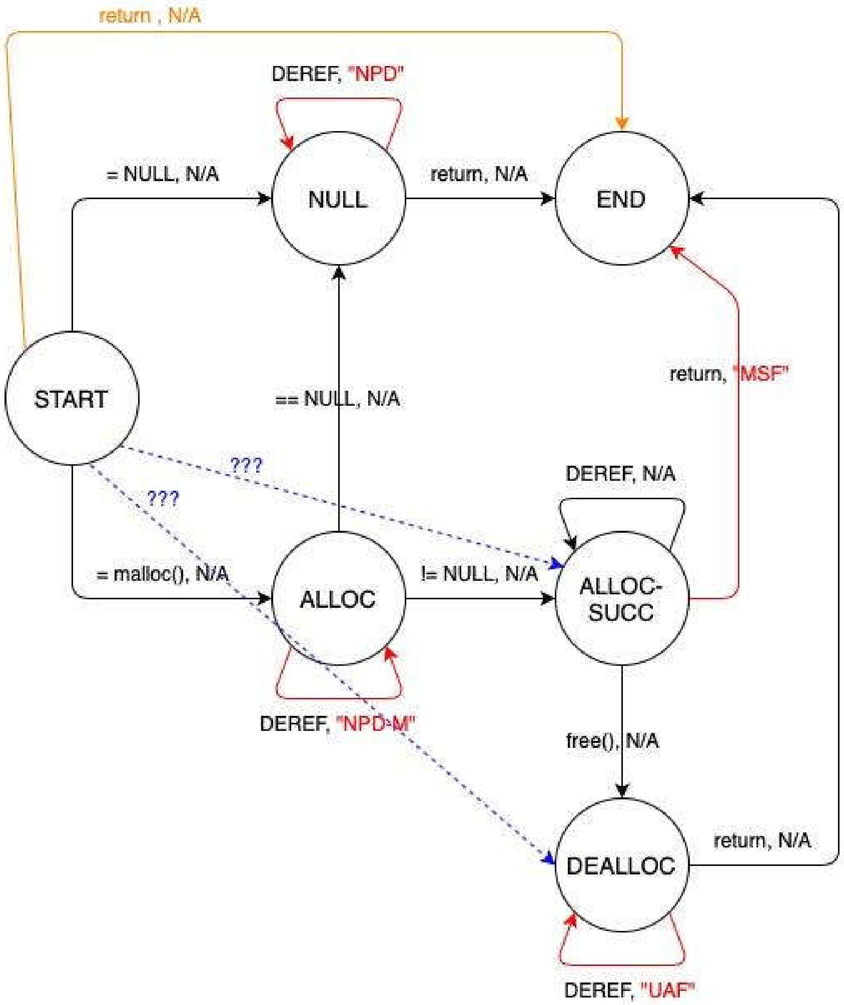
EXAMPLE: NULL POINTER DEREFERENCE FSM ABSTRACTION

Precise enough?



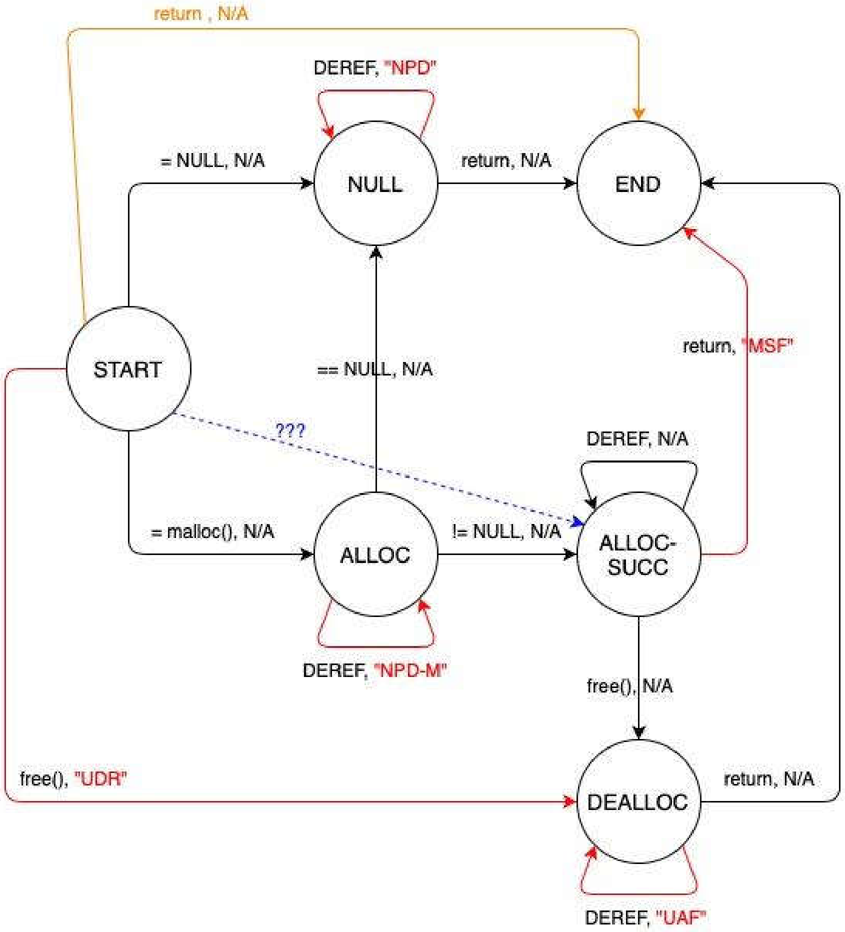
EXAMPLE: NULL POINTER DEREFERENCE FSM ABSTRACTION

No codes related



EXAMPLE: NULL POINTER DEREFERENCE FSM ABSTRACTION

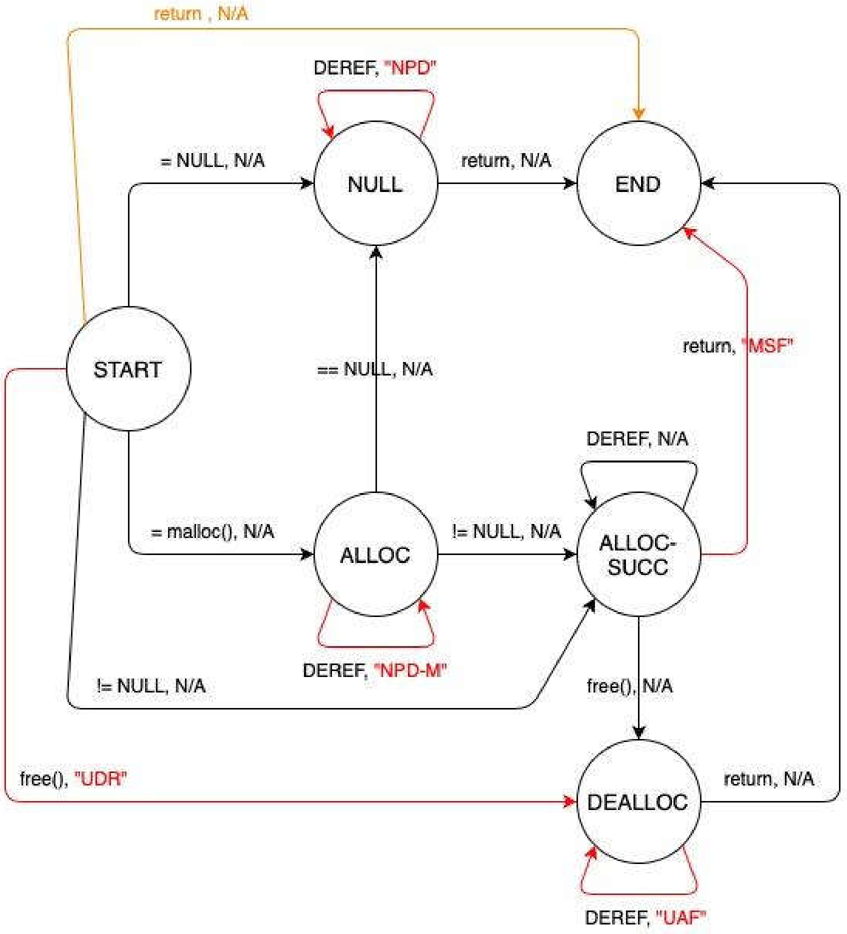
Use Dangling Reference



EXAMPLE: NULL POINTER DEREFERENCE FSM ABSTRACTION

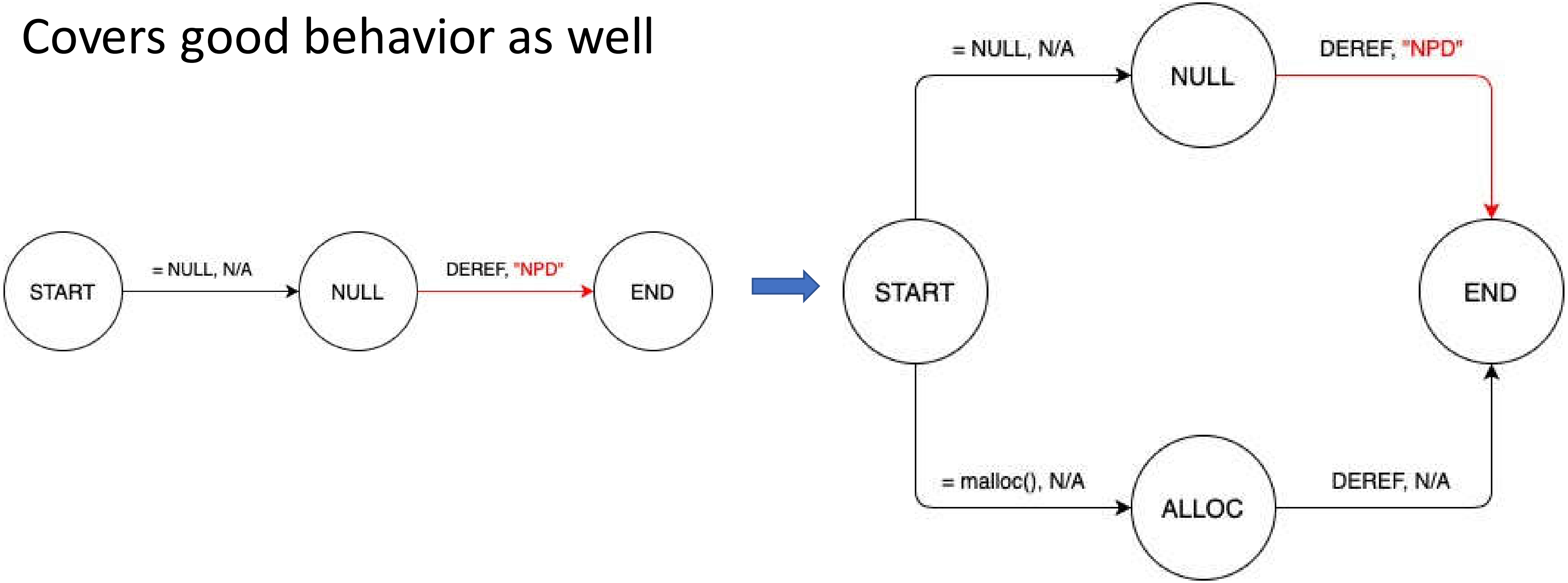
A not-null check can avoid NPD as well

Consider library / SDK APIs



FSM ABSTRACTION BUILD UP

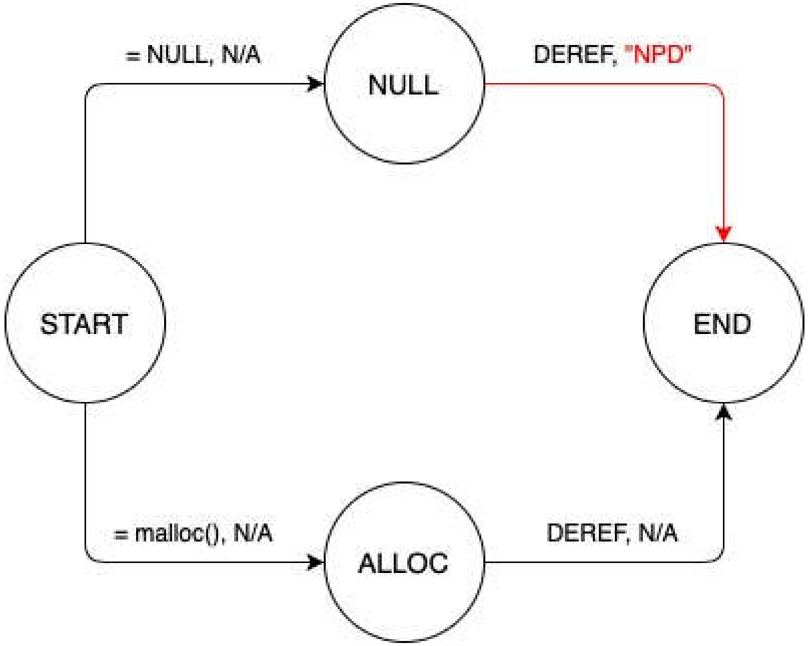
Covers good behavior as well

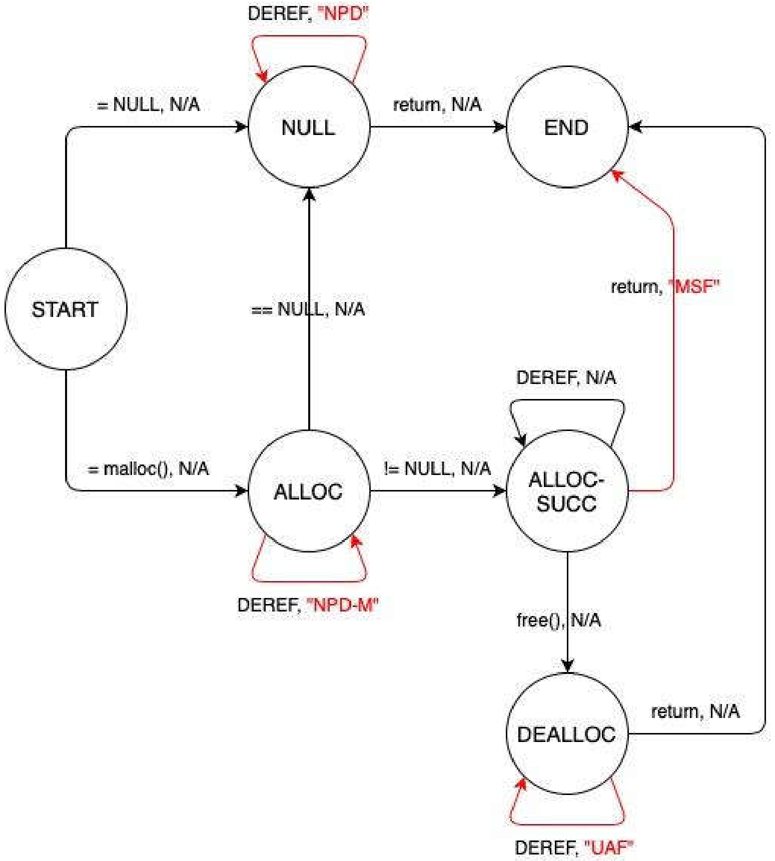


Company Confidential: Please do not distribute.

FSM ABSTRACTION BUILD UP

Considers the side effects

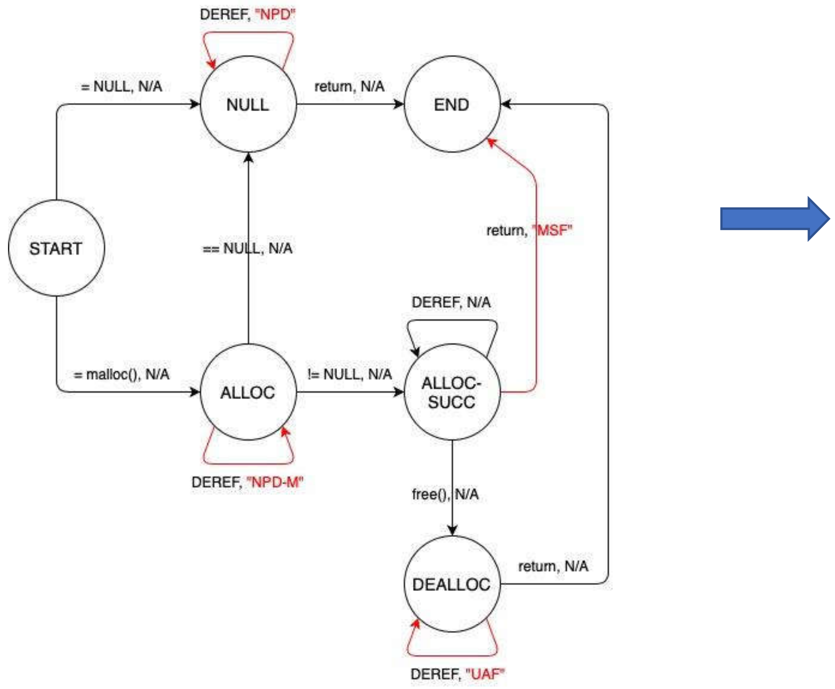


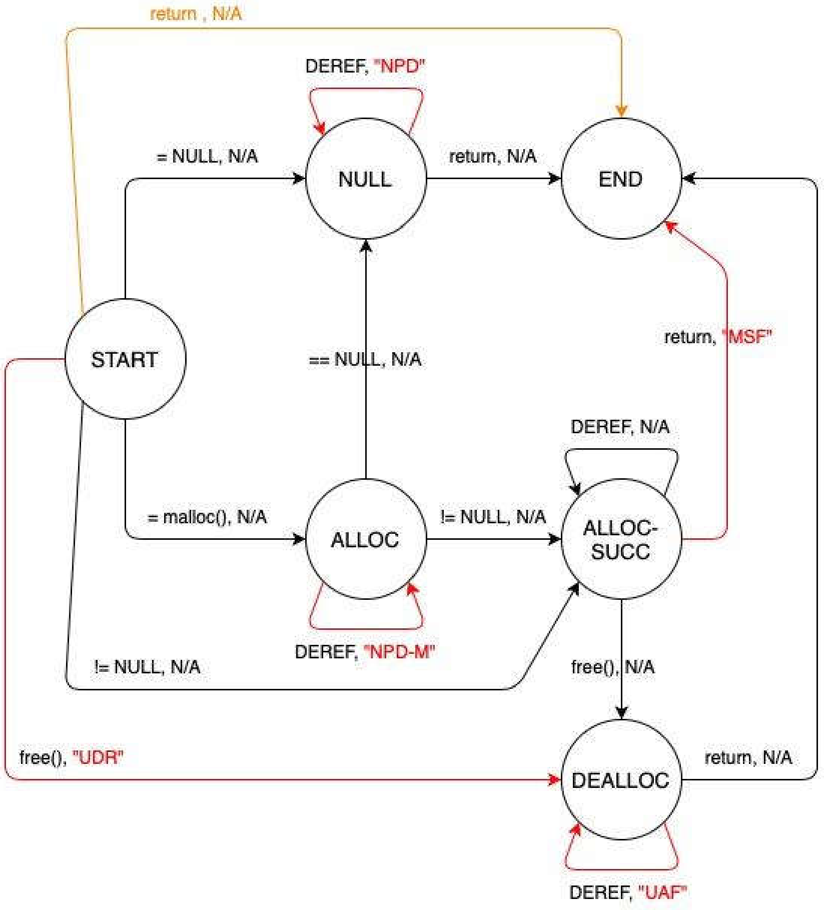


Company Confidential: Please do not distribute.

FSM ABSTRACTION BUILD UP

Tries to reach all other states from one





xcalscan FSM ABSTRACTION BUILD UP

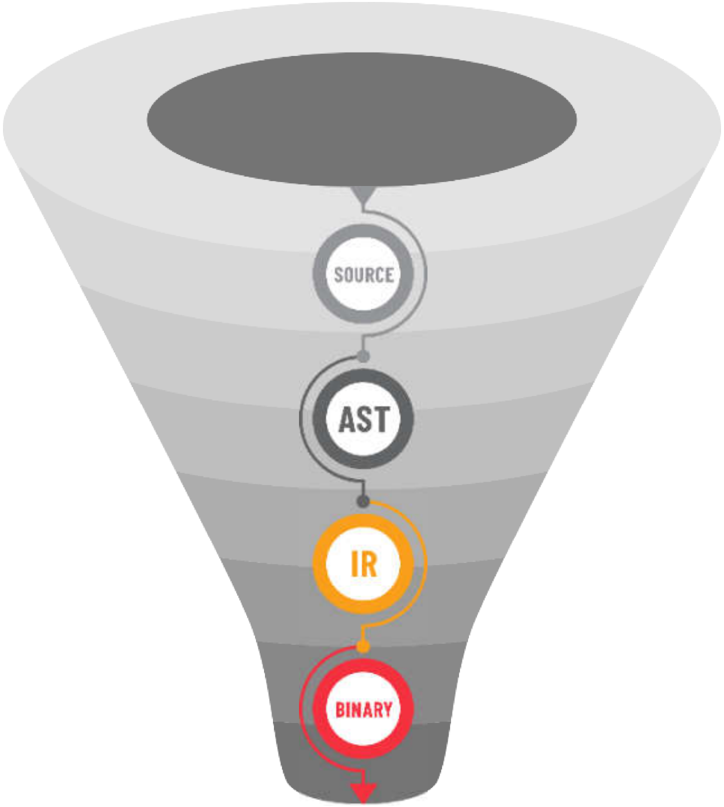
- ✓ Covers good behavior as well
- ✓ Considers the side effects
- ✓ Tries to reach all other states from one
- ✓ Iterates above until a fixed point

xcalscan PERFORMANCE EVALUATION

✓ Analyze based on SSA IRs
  - Context sensitive
  - Flow sensitive
  - Cross file
  - Cross language

✓ On demand analysis
  - Less time
  - Less memory

✓ Symbolic evaluation
  - User customizable rules



xcalscan PERFORMANCE EVALUATION

✓ Analyze based on SSA IRs

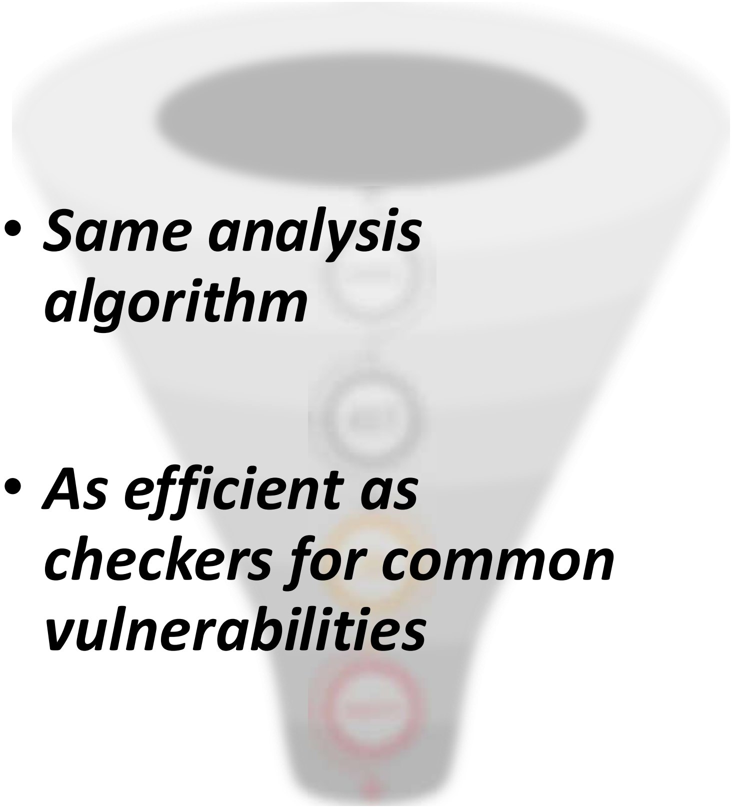
- Context sensitive
- Flow sensitive
- Cross file
- Cross language

✓ On demand analysis

- Less time
- Less memory

✓ Symbolic evaluation

- User customizable rules



- ***Same analysis algorithm***
- ***As efficient as checkers for common vulnerabilities***

EVALUATION: TIME & MEMORY FOOTPRINT

PROJECT	LOC	XCALSCAN*		PINPOINT 1.5**	
		Scan Time	Memory Usage	Scan Time	Memory Usage
MySQL 5.5.10	1M	8m19s	11.2GB	1.5h*	60GB*
OpenSSL 1.0.1f	500K	2m28s	2.9GB	-	-
GDB 7.0a	490K	1m24s	3.5GB	-	-

C: ~3K LOC/s (65 projects, 16,438,505 LOC)

JAVA: ~0.5k LOC /s (9 projects, 736,828 LOC)

** XCALSCAN on CentOS7, 64GB Mem + 17GB Swap, Intel i7-9700 @ 3.00GHz (8 core)*

*** Source: PINPOINT PLDI2018 paper, https://www.cse.ust.hk/~charlesz/pinpoint.pdf*

EVALUATION: ACCURACY

Juliet C	XCALSCAN	CLANG
True Positive (TP, Higher better)	64%	21%
False Positive (FP, Lower better)	6%	14%

✓ Benchmark

- Juliet C has 74699 cases, 118 categories.
- 42 supported, 26 of windows tests excluded

✓ Version

- Clang 3.8.0-2

EVALUATION: ACCURACY CONTINUED

PROJECT	RULE	XCALSCAN				OTHER TOOL			
		Errors reported	True Positive	False Positive	FP Rate	Errors reported	True Positive	False Positive	FP Rate
MySQL 5.5.10	NPD	261	147	114	43%	69	55	14	20%
	UAF	40	22	18	45%	19	12	7	36%
OpenSSL 1.0.1f	NPD	65	46	19	29%	48	42	6	12%
	UAF	18	6	12	66%	6	3	3	50%
GDB 7.0a	NPD	88	50	38	43%	12	11	1	8%
	UAF	15	4	11	73%	5	1	4	80%

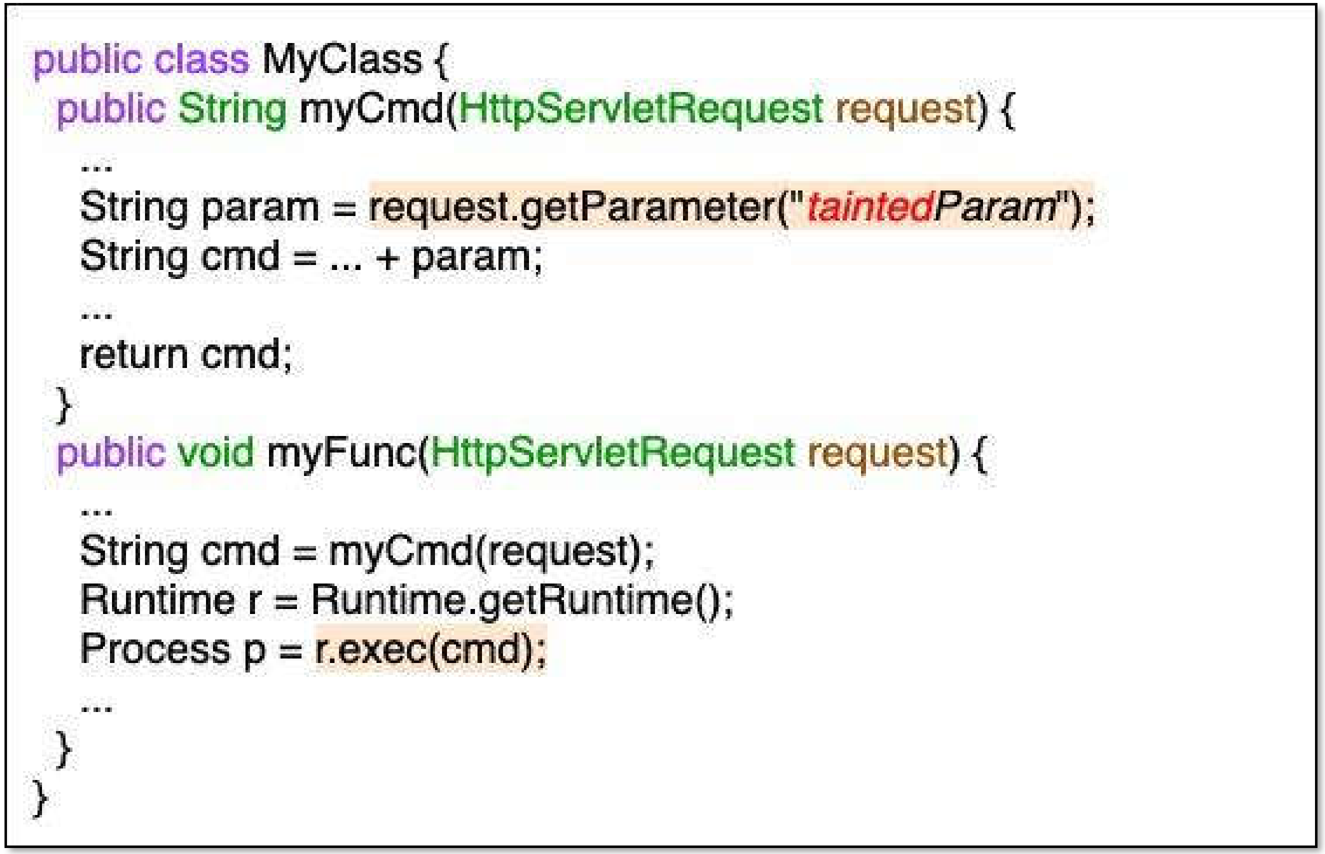
EVALUATION: CROSS PROCEDURE

Juliet C	XCALSCAN	XCALSCAN (cross procedure feature disabled)
True Positive (TP, Higher better)	64%	36%
False Positive (FP, Lower better)	6%	28%

✓ Benefit of cross procedure analysis

- Higher True Positive rate
- Lower False Positive rate

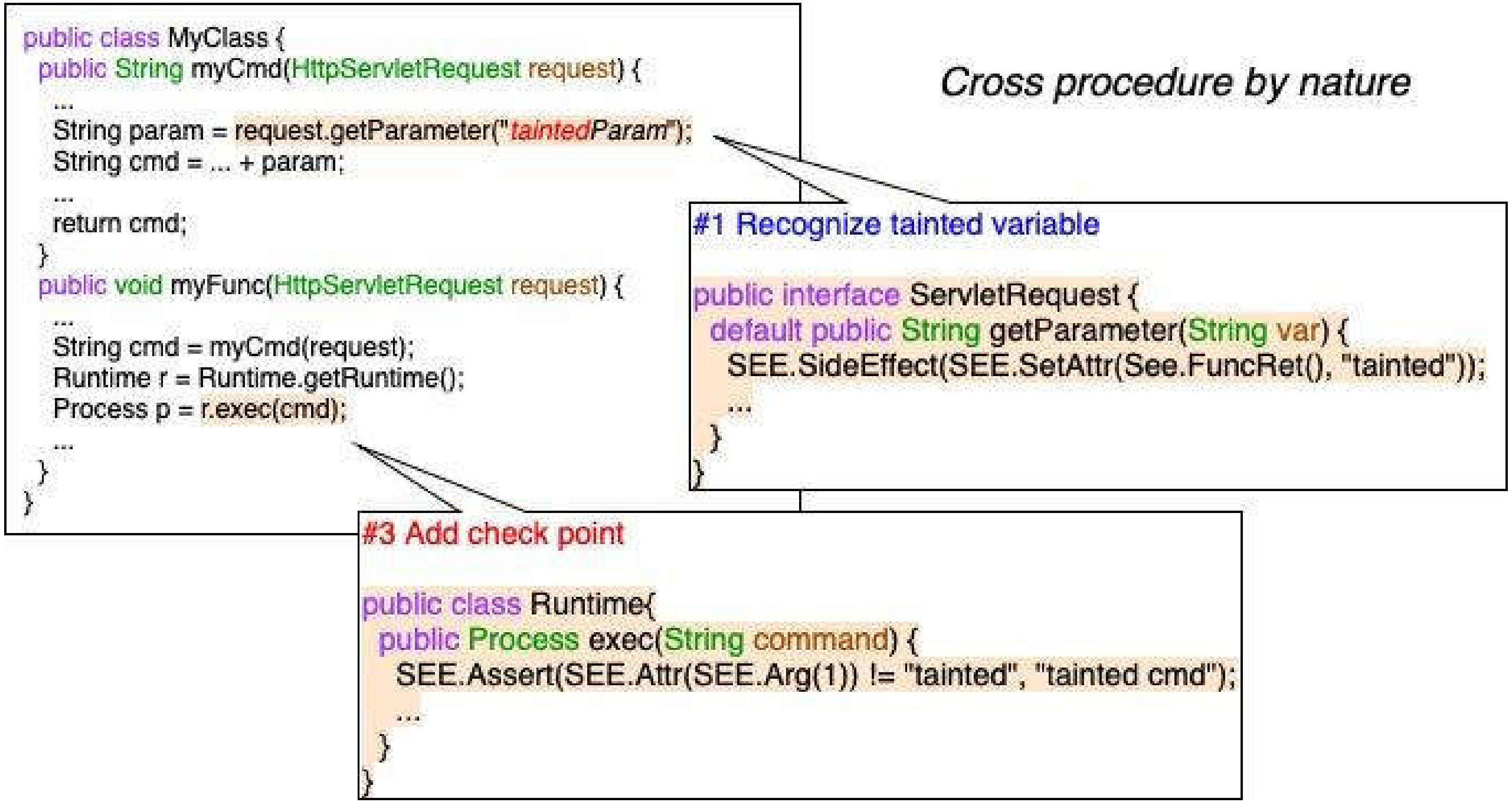
EXAMPLE: INJECTION – CROSS PROCEDURE



```
public class MyClass {
  public String myCmd(HttpServletRequest request) {
    ...
    String param = request.getParameter("taintedParam");
    String cmd = ... + param;
    ...
    return cmd;
  }
  public void myFunc(HttpServletRequest request) {
    ...
    String cmd = myCmd(request);
    Runtime r = Runtime.getRuntime();
    Process p = r.exec(cmd);
    ...
  }
}
```

EXAMPLE: INJECTION – CROSS PROCEDURE

Cross procedure by nature



```
public class MyClass {
  public String myCmd(HttpServletRequest request) {
    ...
    String param = request.getParameter("taintedParam");
    String cmd = ... + param;
    ...
    return cmd;
  }
  public void myFunc(HttpServletRequest request) {
    ...
    String cmd = myCmd(request);
    Runtime r = Runtime.getRuntime();
    Process p = r.exec(cmd);
    ...
  }
}
```

#1 Recognize tainted variable

```
public interface ServletRequest {
  default public String getParameter(String var) {
    SEE.SideEffect(SEE.SetAttr(See.FuncRet(), "tainted"));
    ...
  }
}
```

#3 Add check point

```
public class Runtime{
  public Process exec(String command) {
    SEE.Assert(SEE.Attr(SEE.Arg(1)) != "tainted", "tainted cmd");
    ...
  }
}
```

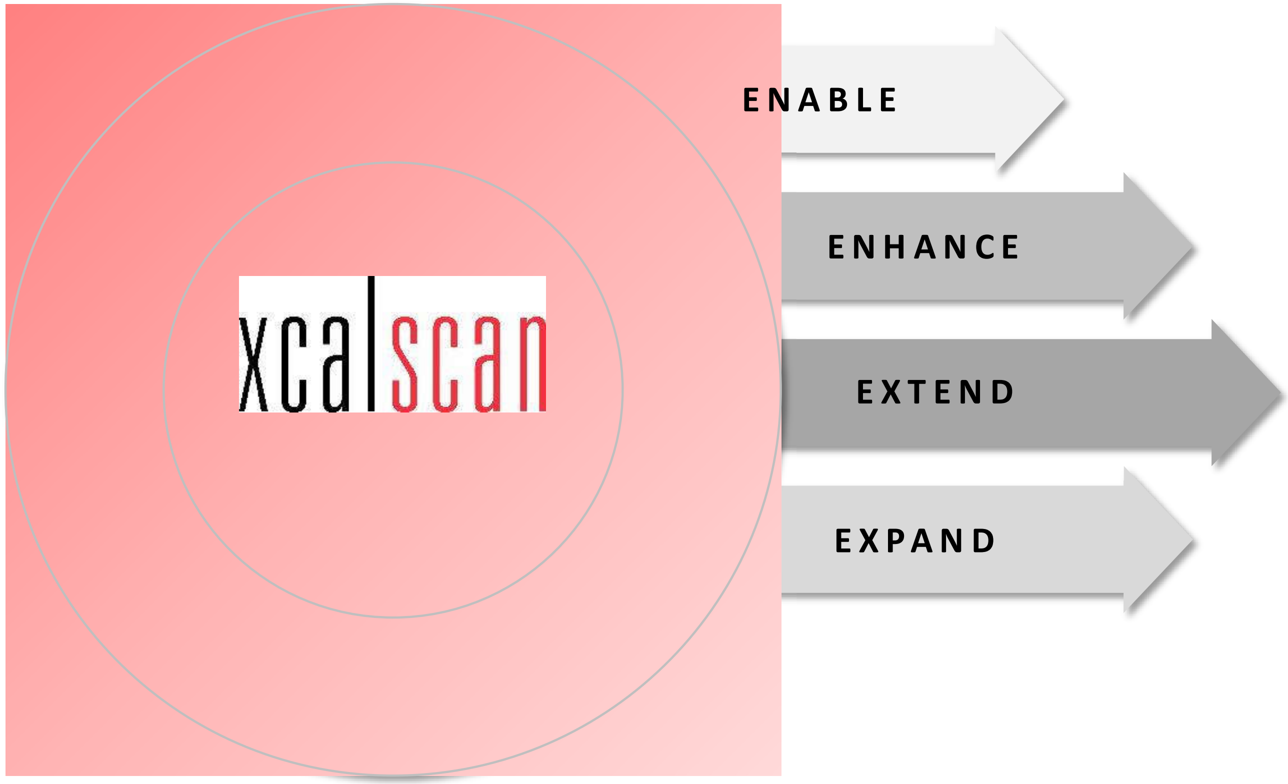
SUMMARY

1 Business Logic verification is critical

2 Symbolic evaluation enables customizability

3 Evaluate SAST by speed, memory consumption, accuracy & cross procedural capability

XCALIBYTE'S FUTURE DIRECTIONS



Enable the framework to work on more resource kinds: lock, thread…

Visual programming tools for rules

Integrated Application Security Testing (IAST)

JavaScript, Python, Rust & other languages

CPP-Summit

Q&A

xcalibyte

鉴释 xcalibyte