



Vulnerability Detection - Best Effort

Holistic Software Security

Aravind Machiry



Best effort techniques

- We don't need to find all the bugs, i.e., we do not need to be sound.
 - “Unsoundness was controversial in the research community, though it has since become almost a de facto tool bias for commercial products and many research projects.”
- Precision (i.e., no false positives is more important):
 - “False positives do matter. In our experience, more than 30% easily cause problems. People ignore the tool.”



Gen 1: User written patterns.

- METAL: Use compiler to check user written patterns.
- Custom language for checker:
 - Hard to write accurate patterns for common developers.
- False positives:
 - No field sensitivity and context sensitivity.

```
sm range_check {  
  // Wild-card variables used in patterns.  
  decl any_expr y, z, len; // match any expr  
  decl any_pointer v;      // match any pointer  
  state decl any_expr x;    // bind state to x  
  
  // Start state. Matches any copy_from_user  
  // call and puts parameter x in tainted state.  
  start: { copy_from_user(x, y, len) }  
    ==> x.tainted  
;  
// Catch operations illegal on unsafe values.  
x.tainted, x.need_ub, x.need_lb:  
  { v[x] } ==> { err("Dangerous index!"); }  
| { copy_from_user(y, z, x) }  
| { copy_to_user(y, z, x) }  
  ==> { err("Dangerous length arg!"); }  
;  
// Named patterns that match upper-bound  
// (ub) and lower-bound checks (lb).  
pat ub = { x < y } | { x <= y };  
pat lb = { x > y } | { x >= y };  
  
;  
x.need_lb:  
  lb ==> true=x.stop, false=x.need_lb  
| ub ==> true=x.need_lb, false=x.stop  
| { x == y } ==> true=x.stop, false=x.need_lb  
| { x != y } ==> true=x.need_lb, false=x.stop  
;  
}
```



Gen 2: Make the patterns easier.

- Microgrammers:
 - Only parse required language features => Custom parsers for each checker.
- To detect Null ptr dereference:
 - We need to parse “if” and pointer-dereference.

- **No need to have full compiler front-end.**

```
S          → Function
Function   → Ident '(' FnArgs SkipTo('{') Body ')'
FnArgs     → SkipTo(',', FnArgs | ')')

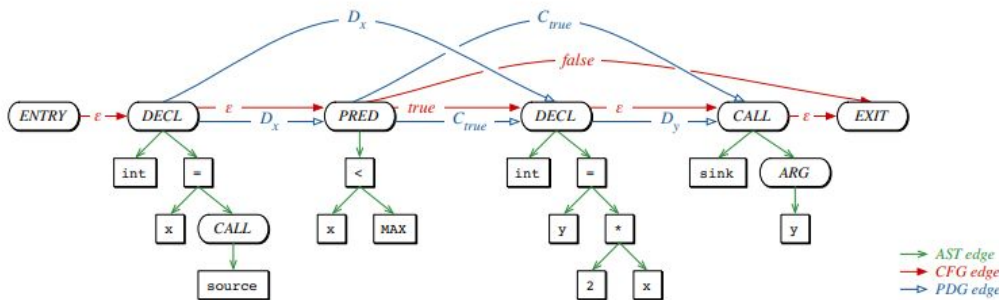
Body       → '{' Statements '}' | Statement
Statements → Statement Statements | ε
Statement  → IfStmt | WhileStmt Body | ... | Line
IfStmt     → if Balanced('(' , ')') Body [Else]
Else       → else Body
...        // more constructs
Line       → SkipTo(';')
```

Gen 3: Make patterns even easier.

- Code Property Graphs:
 - Queries using Gremlin: Graph query over code property graph.

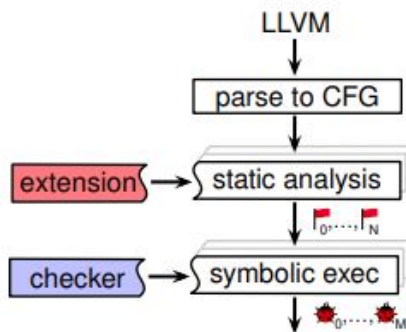
```
void foo()  
{  
  int x = source();  
  if (x < MAX)  
  {  
    int y = 2 * x;  
    sink(y);  
  }  
}
```

1
2
3
4
5
6
7
8
9



Gen 4: Reduce false positives.

- Sys: static/symbolic tool:
 - Use static extensions to find potential flows and then use symbolic checker to filter out false positives.





Sys: Static/Symbolic tool

- Sys: static/symbolic tool:
 - Use static extension to save and report paths and symbolic checker to accurately find the bugs.

Static Extension

```
1 check :: Named Instruction -> Checker ()
2 check instr = case instr of
3
4   -- Save the size of the object
5   name := Call fName args | isAllocation fName -> do
6     let allocSize = args !! 0
7     saveSize name allocSize
8
9   -- Keep track of dependencies between LHS and RHS
10  -- variables of arithmetic instructions.
11  name := _ | isArith instr -> do
12    operands <- getOperands instr
13    forM_ operands $ addDep name
14
15  -- If an array index has some dependency on
16  -- an object's allocated size, report the path
17  name := GetElementPtr addr (arrInd:_) -> do
18    let addrName = nameOf addr
19    addrSize <- findSize addrName
20    when (isDep addrSize arrInd) $
21      reportPath arrSize arrInd
22
23  -- Otherwise do nothing
24  _ -> return ()
```

- False positives



Sys: Static/Symbolic tool

- Sys: static/symbolic tool:
 - Use static extension to save and report paths and symbolic checker to accurately find the bugs.

Static Extension

```
1 check :: Named Instruction -> Checker ()
2 check instr = case instr of
3
4   -- Save the size of the object
5   name := Call fName args | isAllocation fName -> do
6     let allocSize = args !! 0
7     saveSize name allocSize
8
9   -- Keep track of dependencies between LHS and RHS
10  -- variables of arithmetic instructions.
11  name := _ | isArith instr -> do
12    operands <- getOperands instr
13    forM_ operands $ addDep name
14
15  -- If an array index has some dependency on
16  -- an object's allocated size, report the path
17  name := GetElementPtr addr (arrInd:_) -> do
18    let addrName = nameOf addr
19    addrSize <- findSize addrName
20    when (isDep addrSize arrInd) $
21      reportPath arrSize arrInd
22
23  -- Otherwise do nothing
24  _ -> return ()
```

Symbolic Checker

```
1 symexCheck :: Name -> Name -> Symex ()
2 symexCheck arrSize arrInd = do
3
4   -- Turn the size into a symbolic bitvector
5   arrSizeSym <- getName arrSize
6   -- Turn the index into a symbolic bitvector
7   let indTy = typeOf arrInd
8   arrIndSym <- getName arrInd
9   arrIndSize <- toBytes indTy arrInd
10
11  -- Report a bug if the index size can be
12  -- larger than the allocation size
13  assert $ isUge byte arrIndSize arrSizeSym
```



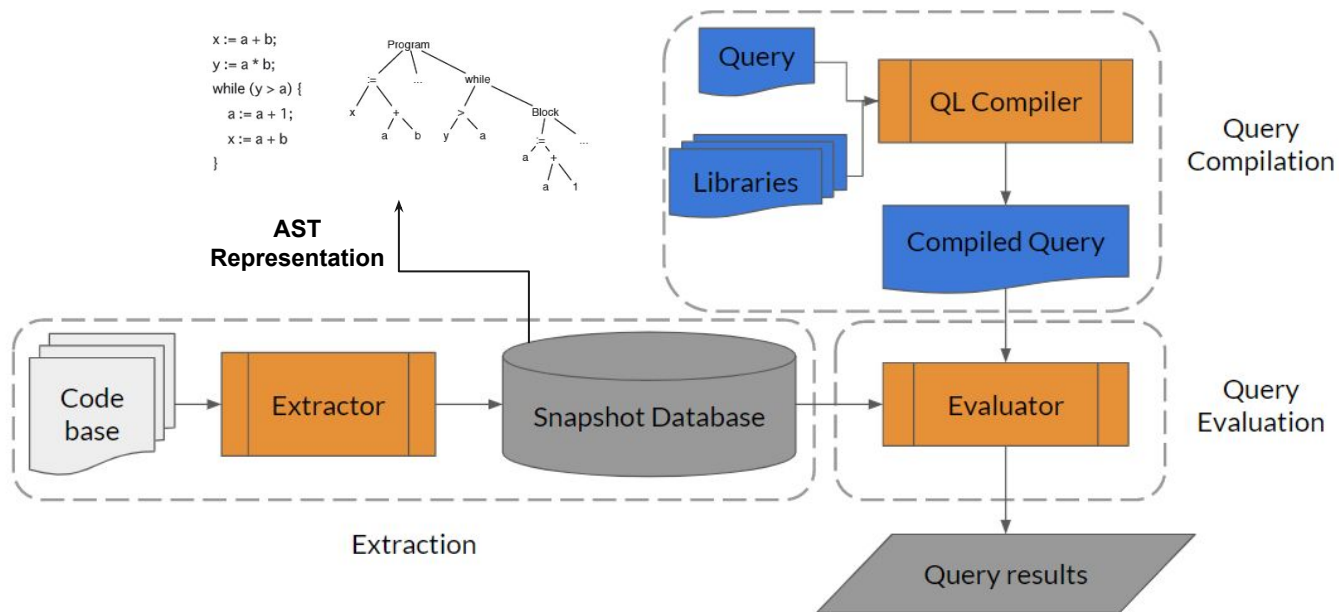

CodeQL: State of the art (Pattern based)

- Logical query language similar to SQL.
- Modular and extensible:
 - Support for custom classes.
 - Libraries.
- Well maintained by Microsoft and its open source.

123 CVEs discovered by GitHub Security Lab (prior to March 2020)

- 🛡 Use-after-free in memory pools during data transfer
[CVE-2020-9273](#) • ProFTPD • published 1 years ago • discovered by [Antonio Morales](#)
- 🛡 OOB Read in `getstateflags` function
[CVE-2020-9272](#) • ProFTPD • published 1 years ago • discovered by [Antonio Morales](#)
- 🛡 Multiple int-to-bool casting vulnerabilities, leading to heap overflow
[CVE-2020-6835](#) • bftpd Bftpd • published 1 years ago • discovered by [Antonio Morales](#)
- 🛡 OOB read in `bftpd` due to uninitialized value in `hidegroups_init()` function
[CVE-2020-6162](#) • bftpd Bftpd • published 1 years ago • discovered by [Antonio Morales](#)
- 🛡 Potential buffer overflow in `ModPlug_SampleName` and `ModPlug_InstrumentName`
[CVE-2019-17113](#) • OpenMPT libopenmpt • published 1 years ago • discovered by [Antonio Morales](#)

CodeQL





CodeQL : Example

```
void fire_thrusters(double vectors[12]) {  
    for (int i = 0; i < 12; i++) {  
        ... vectors[i] ...  
    }  
}  
  
...  
  
double thruster[3] = ... ;  
  
fire_thrusters(thruster);
```

- What is the problem with the code?



CodeQL : Example

```
void fire_thrusters(double vectors[12]) {  
    for (int i = 0; i < 12; i++) {  
        ... vectors[i] ...  
    }  
}  
  
...  
  
double thruster[3] = ... ;  
  
fire_thrusters(thruster);
```

- What is the problem with the code?
- In C, array types of parameters degrade to pointer types.
- The size is ignored!



Finding using CodeQL

```
void fire_thrusters(double vectors[12]) {  
    for (int i = 0; i < 12; i++) {  
        ... vectors[i] ...  
    }  
}  
  
...  
  
double thruster[3] = ... ;  
  
fire_thrusters(thruster);
```

- Basic Query Structure:

import <language library>

from <entities and types>

where <queries on entities>

select <entities to select>



Finding using CodeQL

```
void fire_thrusters(double vectors[12]) {  
    for (int i = 0; i < 12; i++) {  
        ... vectors[i] ...  
    }  
}  
  
...  
  
double thruster[3] = ... ;  
  
fire_thrusters(thruster);
```

- First, let's find function calls.

```
import cpp  
  
from Function f, FunctionCall c  
  
where f = c.getTarget()  
  
select "Found call to " + f.getName()
```



Finding using CodeQL

```
void fire_thrusters(double vectors[12]) {  
    for (int i = 0; i < 12; i++) {  
        ... vectors[i] ...  
    }  
}  
  
...  
  
double thruster[3] = ... ;  
  
fire_thrusters(thruster);
```

- Argument should be constant array.

```
import cpp  
  
from Function f, FunctionCall c, int i  
  
where f = c.getTarget()  
  
and (f.getParameter(i).getType() instanceof ArrayType)  
  
select "Found call to " + f.getName()
```



Finding using CodeQL

```
void fire_thrusters(double vectors[12]) {  
    for (int i = 0; i < 12; i++) {  
        ... vectors[i] ...  
    }  
}  
  
...  
  
double thruster[3] = ... ;  
  
fire_thrusters(thruster);
```

Checking size of parameter and argument:

```
import cpp  
  
from Function f, FunctionCall c, int i, int a, int b  
  
where f = c.getTarget()  
  
and a = c.getArgument(i).getType().(ArrayType).getArraySize()  
  
and b = f.getParameter(i).getType().(ArrayType).getArraySize()  
  
and a < b  
  
select c.getArgument(i), f, "Found vulnerability"
```


Finding using CodeQL

```
void fire_thrusters(double vectors[12]) {  
    for (int i = 0; i < 12; i++) {  
        ... vectors[i] ...  
    }  
}  
  
...  
  
double thruster[3] = ... ;  
  
fire_thrusters(thruster);
```

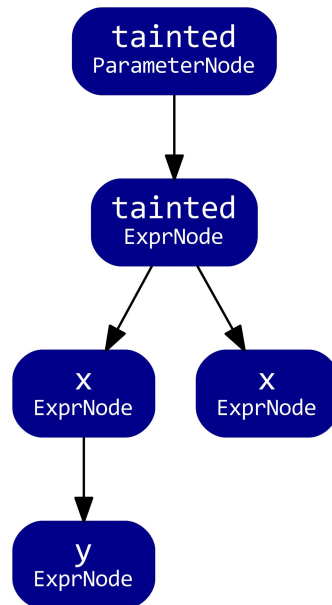
Checking size of parameter and argument:

Characteristic predicate

```
import cpp  
  
from Function f, FunctionCall c, int i, int a, int b  
  
where f = c.getTarget()  
  
and a = c.getArgument(i).getType().(ArrayType).getArraySize()  
  
and b = f.getParameter(i).getType().(ArrayType).getArraySize()  
  
and a < b  
  
select c.getArgument(i), f, "Found vulnerability"
```

CodeQL : Data flow analysis

```
int func(int tainted) {  
    int x = tainted;  
    if (someCondition) {  
        int y = x;  
        callFoo(y);  
    } else {  
        return x;  
    }  
    return -1;  
}
```





CodeQL : Data flow example

Format string checker: Use of externally controlled format string.

```
printf(userControlledString, arg1);
```

Goal: Find uses of printf (or similar) where the format string can be controlled by an attacker.



CodeQL : format string checker

```
import cpp

import semmle.code.cpp.dataflow.DataFlow
import semmle.code.cpp.common.Printf

class SourceNode extends DataFlow::Node {
  SourceNode() {
    not DataFlow::localFlowStep(_, this)
  }
}

from FormattingFunction f, Call c, SourceNode src, DataFlow::Node arg
where c.getTarget() = f and arg.asExpr() = c.getArgument(f.getFormatParameterIndex()) and
      DataFlow::localFlow(src, arg) and not src.asExpr() instanceof StringLiteral
select arg, "Non-constant format string."
```



CodeQL : format string checker

```
import cpp
import semmlc.code.cpp.dataflow.DataFlow
import semmlc.code.cpp.common.Printf
class SourceNode extends DataFlow::Node {
  SourceNode() {
    not DataFlow::localFlowStep(_, this)
  }
}

from FormattingFunction f, Call c, SourceNode src, DataFlow::Node arg
where c.getTarget() = f and arg.asExpr() = c.getArgument(f.getFormatParameterIndex()) and
      DataFlow::localFlow(src, arg) and not src.asExpr() instanceof StringLiteral
select arg, "Non-constant format string."
```

Source node in data flow.

CodeQL : format string checker

```
import cpp
import semmle.code.cpp.dataflow.DataFlow
import semmle.code.cpp.common.Printf
class SourceNode extends DataFlow::Node {
  SourceNode() {
    not DataFlow::localFlowStep(_, this)
  }
}
from FormattingFunction f, Call c, SourceNode src, DataFlow::Node arg
where c.getTarget() = f and arg.asExpr() = c.getArgument(f.getFormatParameterIndex()) and
  DataFlow::localFlow(src, arg) and not src.asExpr() instanceof StringLiteral
select arg, "Non-constant format string."
```

Source node in data flow.

Flows from src to arg.

CodeQL : format string checker

```
import cpp
import semmle.code.cpp.dataflow.DataFlow
import semmle.code.cpp.common.Printf
class SourceNode extends DataFlow::Node {
  SourceNode() {
    not DataFlow::localFlowStep(_, this)
  }
}
from FormattingFunction f, Call c, SourceNode src, DataFlow::Node arg
where c.getTarget() = f and arg.asExpr() = c.getArgument(f.getFormatParameterIndex()) and
  DataFlow::localFlow(src, arg) and not src.asExpr() instanceof StringLiteral
select arg, "Non-constant format string."
```

Source node in data flow.

Flows from src to arg.

Src is not a constant string.



CodeQL: Other features

- Taint analysis:
 - Custom taint source functions and taint sinks.
- LGTM -> Can run various checkers on open source codebases.
- Also, supports various other languages: C++, Java, JavaScript, etc.



CodeQL: Drawbacks

- AST representation (v/s IR representation):
 - Depends on coding pattern.
- Not suitable for complex analysis involving fine-grained data flow.



Pattern based methods

- Fine-grained patterns (e.g., CodeQL, Sys):
 - Hard to write => Need skill.
 - Require build system.
 - Precise.
- Coarse-grained patterns (e.g., Microgrammers, Metal):
 - Relatively easy to write.
 - Does not depend on build system (i.e., no compilation needed).
 - False positives.



Can we automatically generate fine-grained pattern (maybe with small help from developer)?



Best effort: Machine learning

- Category 1: Learn vulnerable code patterns from given dataset.
- Category 2: Bugs as a deviant behavior => Less frequently used pattern most probably suggests a bug.
- Category 3: Assist developers and other techniques to focus on potentially vulnerable code.
- Category 4: Active learning => Interact with users to know how a bug should look like.
- Category 5: Lets use LLMs.



General Principle of ML based Detectors

```
#!/bin/bash

# Verifie la validité d'une date
function dateValide()
{
    local a=`echo $1 | grep ^[0-3][0-9]\|[0-1][0-9]\|[0-9]{4}$`
    if [ "$a" != 0 ];then
        j=`echo $1 | cut -d'/' -f1`
        m=`echo $1 | cut -d'/' -f2`
        a=`echo $1 | cut -d'/' -f3`

        [ $j -le 31 -a $j -ge 1 -a $m -le 12 -a $m -ge 1 ]
        echo $?
    else
        echo 1
    fi
}

# Verifie si une plage horaire $1 est bien comprise dans la plage horaire $2
function appartientPlage()
{
    if [ "`valideHoraire $1`" == "0" -a "`valideHoraire $2`" == "0" ];then
        local h1=`echo $1 | cut -d'-' -f1 | sed s/://`
        local h2=`echo $1 | cut -d'-' -f2 | sed s/://`
        local lim1=`echo $2 | cut -d'-' -f1 | sed s/://`
```

General Principle of ML based Detectors

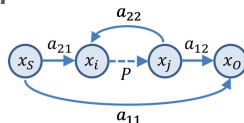
```
#!/bin/bash

# Verifie la validite d'une date
function datevalide()
{
    local _m= echo $1 | grep ^([0-31][0-9]\|([0-31][0-9]\|([0-91(4)5'
    if [ "$m" != 0 ];then
        _j= echo $1 | cut -d'/' -f1
        _m= echo $1 | cut -d'/' -f2
        _s= echo $1 | cut -d'/' -f3
        [ $j -le 31 -a $j -ge 1 -a $m -le 12 -a $s -ge 1 ]
        echo $?
    else
        echo 1
    fi
}

# Verifie si une plage horaire $1 est bien comprise dans la plage horaire $2
function appartientPlage()
{
    if [ "$validHoraire $1" == "0" -a "$validHoraire $2" == "0" ];then
        local h1= echo $1 | cut -d'-' -f1 | sed s/://
        local h2= echo $1 | cut -d'-' -f2 | sed s/://
        local lin1= echo $2 | cut -d'-' -f1 | sed s/://

```

Source Representation



AST, CFG, Program
Dependency Graph (PDG),
Symbolic Trace, etc.

General Principle of ML based Detectors

```
#!/bin/bash

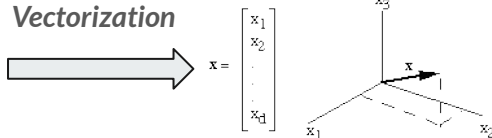
# Verifie la validite d'une date
function datevalide()
{
    local _m= echo $1 | grep ^([0-31][0-9])/([0-31][0-9])/([0-91](4)5)$
    if [ "$_m" != "" ];then
        _m=echo $1 | cut -d'/' -f1
        _m=echo $1 | cut -d'/' -f2
        _m=echo $1 | cut -d'/' -f3
        [ $1 -le 31 -a $2 -le 12 -a $3 -le 31 ]
        echo $?
    else
        echo 1
    fi
}

# Verifie si une plage horaire $1 est bien comprise dans la plage horaire $2
function appartientPlage()
{
    if [ "$validHoraire $1" == "0" -a "$validHoraire $2" == "0" ];then
        local h1=echo $1 | cut -d'-' -f1 | sed s/://
        local h2=echo $1 | cut -d'-' -f2 | sed s/://
        local lin1=echo $2 | cut -d'-' -f1 | sed s/://
    fi
}
```

Source Representation



Vectorization



AST, CFG, Program
Dependency Graph (PDG),
Symbolic Trace, etc.

Bag of words, Word2Vec,
Custom Boolean vector, etc

General Principle of ML based Detectors

```
#!/bin/bash
# Verifie la validite d'une date
function datevalide()
{
    local _m= echo $1 | grep ^[0-31][0-9]\/[0-31][0-9]\/[0-91(4)5]$
    if [ "$_m" != 0 ];then
        _j= echo $1 | cut -d'/' -f1
        _m= echo $1 | cut -d'/' -f2
        _a= echo $1 | cut -d'/' -f3
        [ $_j -le 31 -a $_j -ge 1 -a $_m -le 12 -a $_m -ge 1 ]
        echo $_j
    else
        echo 1
    fi
}

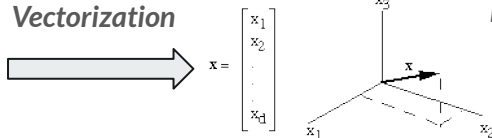
# Verifie si une plage horaire $1 est bien comprise dans la plage horaire $2
function appartientPlage()
{
    if [ "$validetHoraire $1" = "0" -a "$validetHoraire $2" = "0" ];then
        local h1= echo $1 | cut -d'-' -f1 | sed s/://
        local h2= echo $1 | cut -d'-' -f2 | sed s/://
        local l1= echo $2 | cut -d'-' -f1 | sed s/://
    fi
}
```

Source Representation



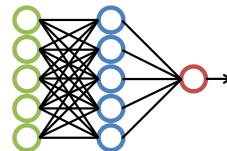
AST, CFG, Program
Dependency Graph (PDG),
Symbolic Trace, etc.

Vectorization



Bag of words, Word2Vec,
Custom Boolean vector, etc

Model Training



RNN, LSTM, SVM,
Clustering, etc.



Category 1: μ VulDeePecker

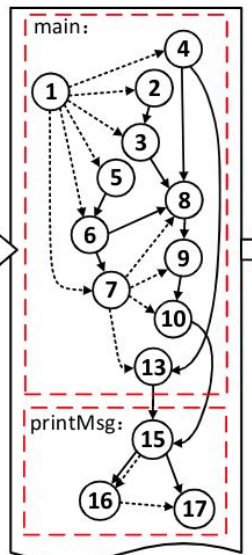
```
1: void main(){
2:   char data[5] = "AAAA";
3:   data[4] = '\0';
4:   char dest[5] = "";
5:   int n;
6:   scanf("%d", &n);
7:   if(n <= 4 && n > 0){
8:     strncpy(dest, data, n);
9:     dest[n] = '\0';
10:    printMsg(dest);
11:  }
12:  else
13:    printMsg(dest);
14: }
15: void printMsg(char* msg){
16:   if(msg != NULL)
17:     printf("%s\n", msg);
18: }
```

Program source code

Category 1: μ VulDeePecker

```
1: void main(){
2:   char data[5] = "AAAA";
3:   data[4] = '\0';
4:   char dest[5] = "";
5:   int n;
6:   scanf("%d", &n);
7:   if(n <= 4 && n > 0){
8:     strncpy(dest, data, n);
9:     dest[n] = '\0';
10:    printMsg(dest);
11:  }
12:  else
13:    printMsg(dest);
14: }
15: void printMsg(char* msg){
16:   if(msg != NULL)
17:     printf("%s\n", msg);
18: }
```

Program source code



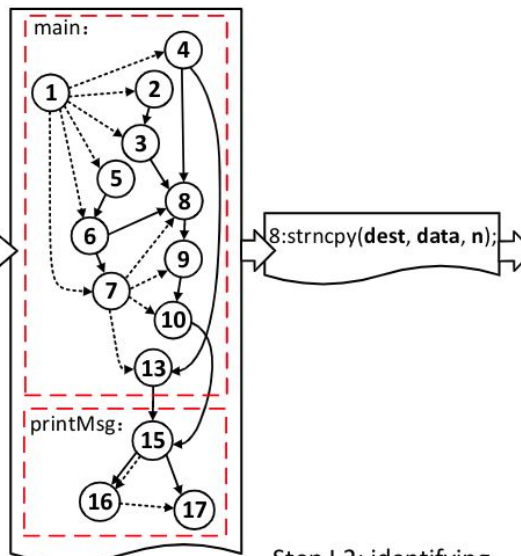
Step 1.1: generating
SDGs

Category 1: μ VulDeePecker

```

1: void main(){
2:     char data[5] = "AAAA";
3:     data[4] = '\0';
4:     char dest[5] = "";
5:     int n;
6:     scanf("%d", &n);
7:     if(n <= 4 && n > 0){
8:         strncpy(dest, data, n);
9:         dest[n] = '\0';
10:        printMsg(dest);
11:    }
12:    else
13:        printMsg(dest);
14: }
15: void printMsg(char* msg){
16:     if(msg != NULL)
17:         printf("%s\n", msg);
18: }

```

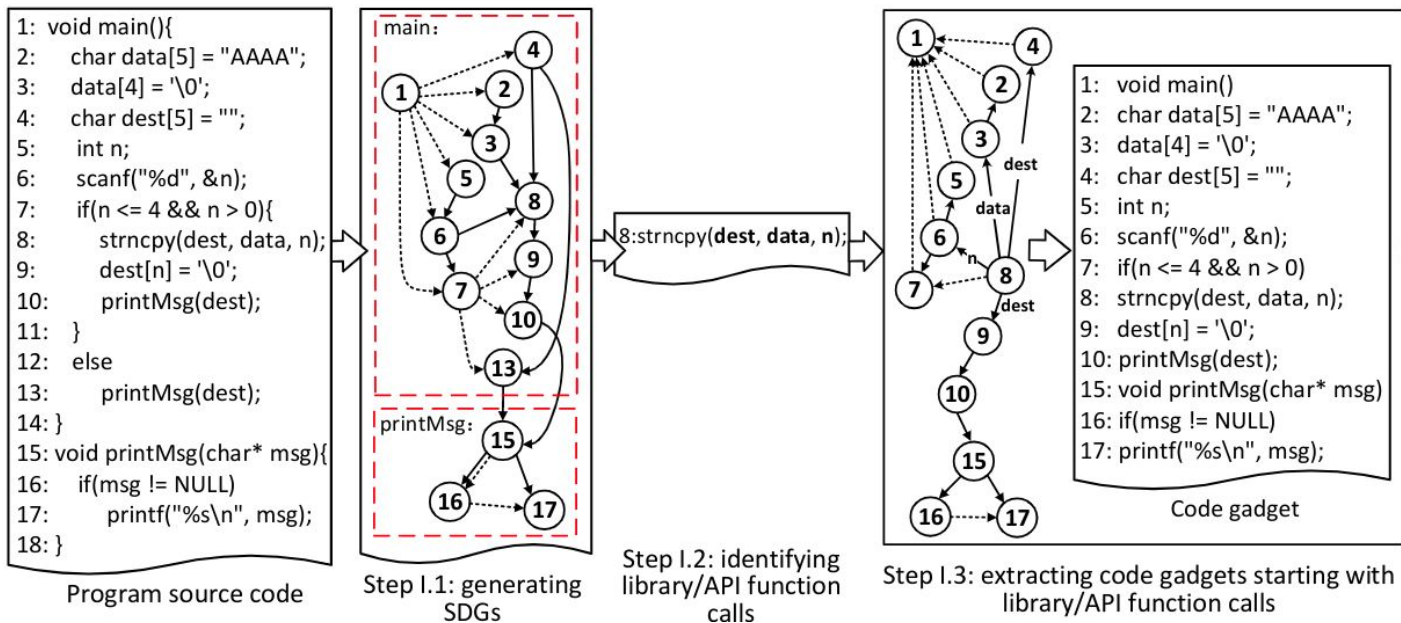


Program source code

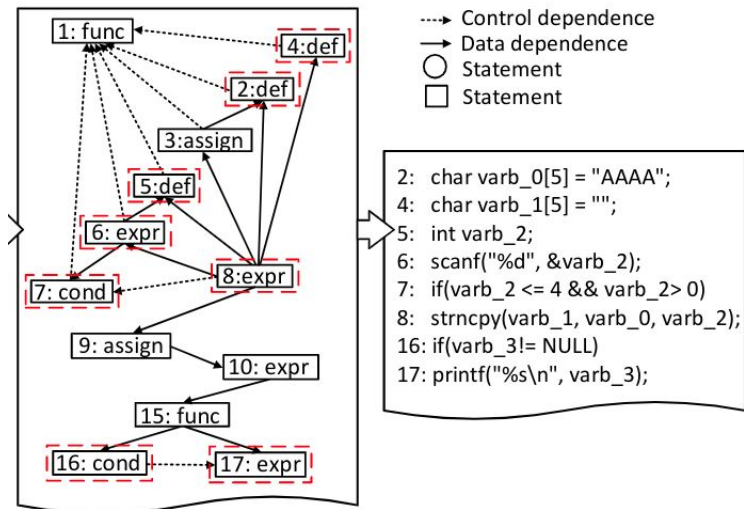
Step I.1: generating SDGs

Step 1.2: identifying library/API function calls

Category 1: μ VulDeePecker



Category 1: μ VulDeePecker



Step IV: generating code attentions corresponding to code gadgets

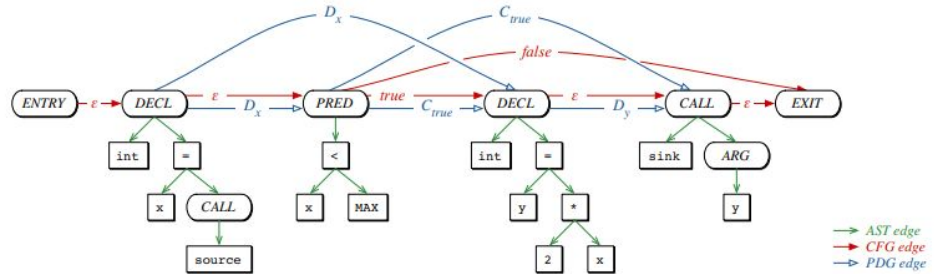
(b) generating code attentions

Category 2: Automatic Inference of Taint Style Vulnerabilities

- Based on Code Property Graphs.

```
void foo()  
{  
  int x = source();  
  if (x < MAX)  
  {  
    int y = 2 * x;  
    sink(y);  
  }  
}
```

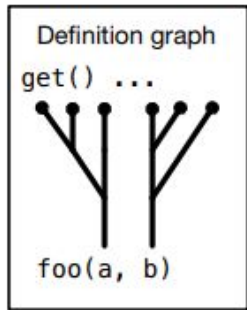
1
2
3
4
5
6
7
8
9



- For each sensitive sink -> Infer the “most common way” the arguments are passed -> create an inverse pattern of the most common way.

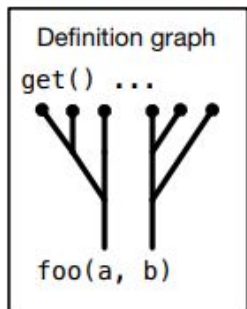
Category 2: Automatic Inference of Taint Style Vulnerabilities

Data-flow analysis

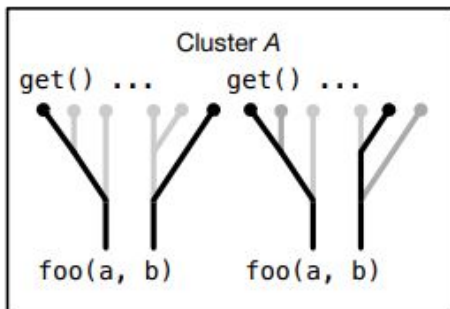


Category 2: Automatic Inference of Taint Style Vulnerabilities

Data-flow analysis

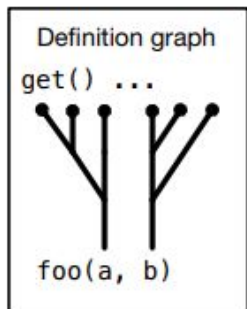


Decompression and clustering

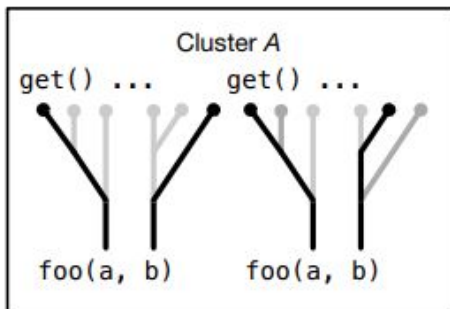


Category 2: Automatic Inference of Taint Style Vulnerabilities

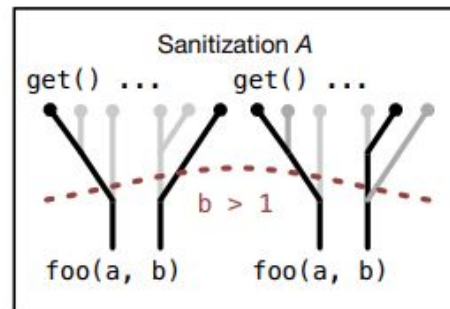
Data-flow analysis



Decompression and clustering

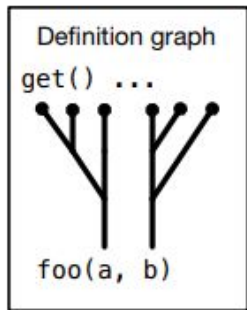


Sanitization overlay

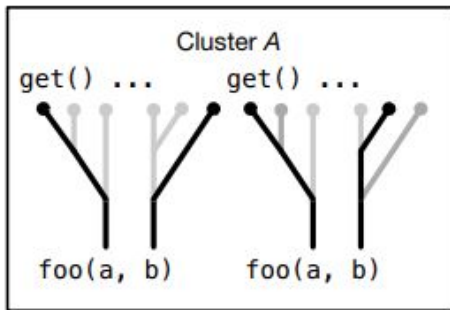


Category 2: Automatic Inference of Taint Style Vulnerabilities

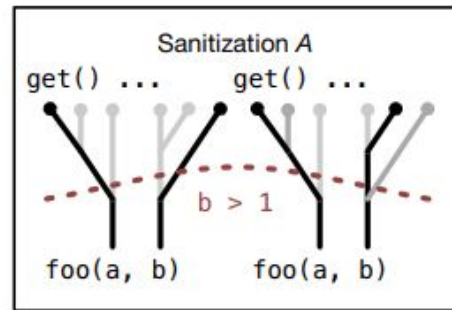
Data-flow analysis



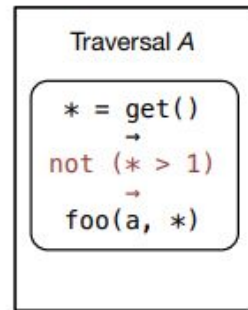
Decompression and clustering



Sanitization overlay



Search patterns






Category 3: Assist Developers in finding Potentially vulnerable code

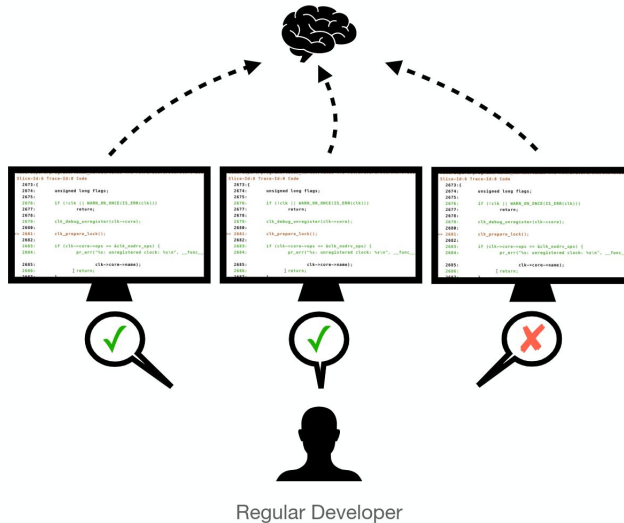
- BRAN: Finds potentially vulnerable functions:
 - Can direct the attention of developer or other tools (E.g., fuzzing) towards these functions.
- Uses representation based features and commit metadata:
 - Size of function, number of pointer accesses, Loops, etc.
 - Number of commits to the function, Reputation of the developer, etc.
- Trains RandomForest to detect vulnerable functions.

Category 4: Active learning

- Existing tools require **a lot of effort by the developers.**
 - Write precise patterns (CodeQL)
 - Hard -> GitHub pays money to write a good CodeQL query.
1. Write a [CodeQL](#) query that models a vulnerability you're interested in.
 2. Run your query on real world open source software and find at least four vulnerabilities, preferably across multiple projects. Please note that projects which purposely include a vulnerability pattern for testing purposes are considered out of scope.
 3. Report the vulnerabilities to the projects' maintainers, help them fix them, and have them obtain CVEs for each one. Remember that for most open source projects, maintainers can now get a CVE directly from GitHub via [Security Advisories](#). **To be eligible for a bounty, you must first coordinate disclosure of the vulnerabilities with the maintainers of the projects.**
 4. Open a pull request [in the CodeQL repo](#) with your CodeQL query. See the [contribution guidelines](#) for more details.
 5. Create an issue using [the bug slayer template](#). The issue should link to your pull request and contain a detailed report of the vulnerabilities your query finds. **Mention only the vulnerabilities that have been publicly disclosed and fixed.** It should include a description of the vulnerabilities, their associated CVEs, and how the query allowed you to find them. Pull requests without an accompanying issue cannot be considered.
 6. An award of up to **\$5000 USD** will be granted  based on the impact and risk associated with the vulnerability and the quality of your query when determining the award amount.

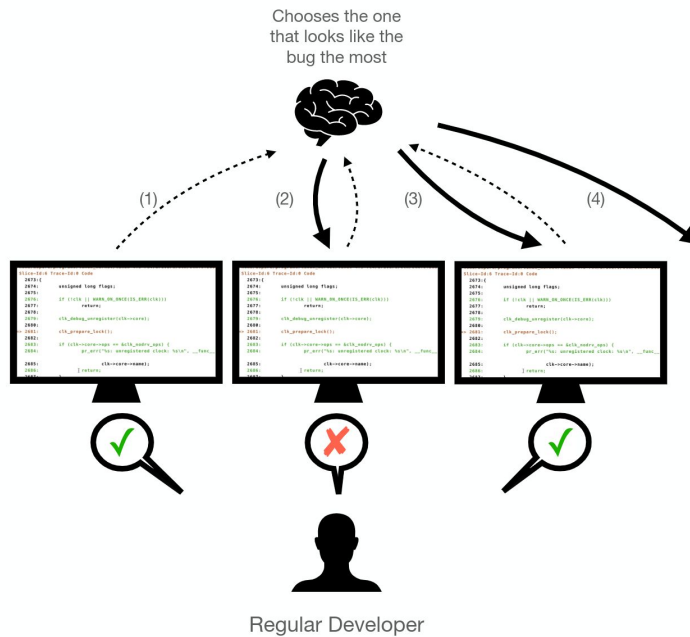
Category 4: Active learning

- What do developers know?
 - Given a warning, they can say whether the warning is true or not.

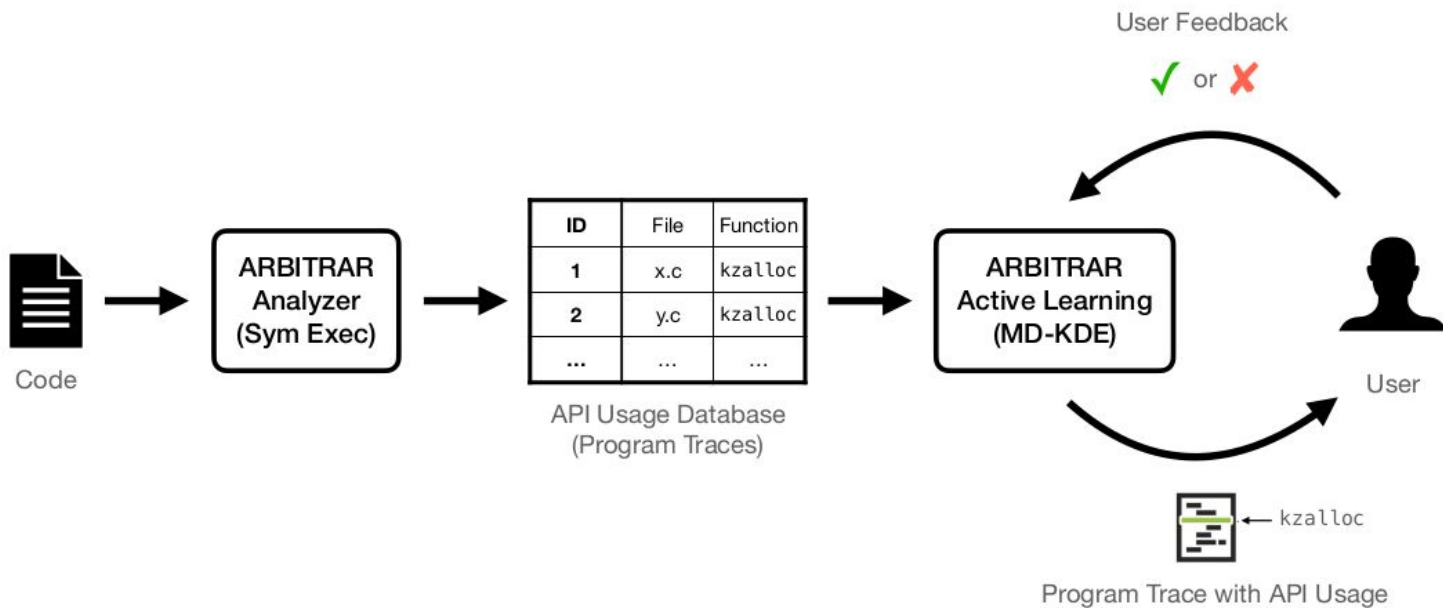


Category 4: Active learning

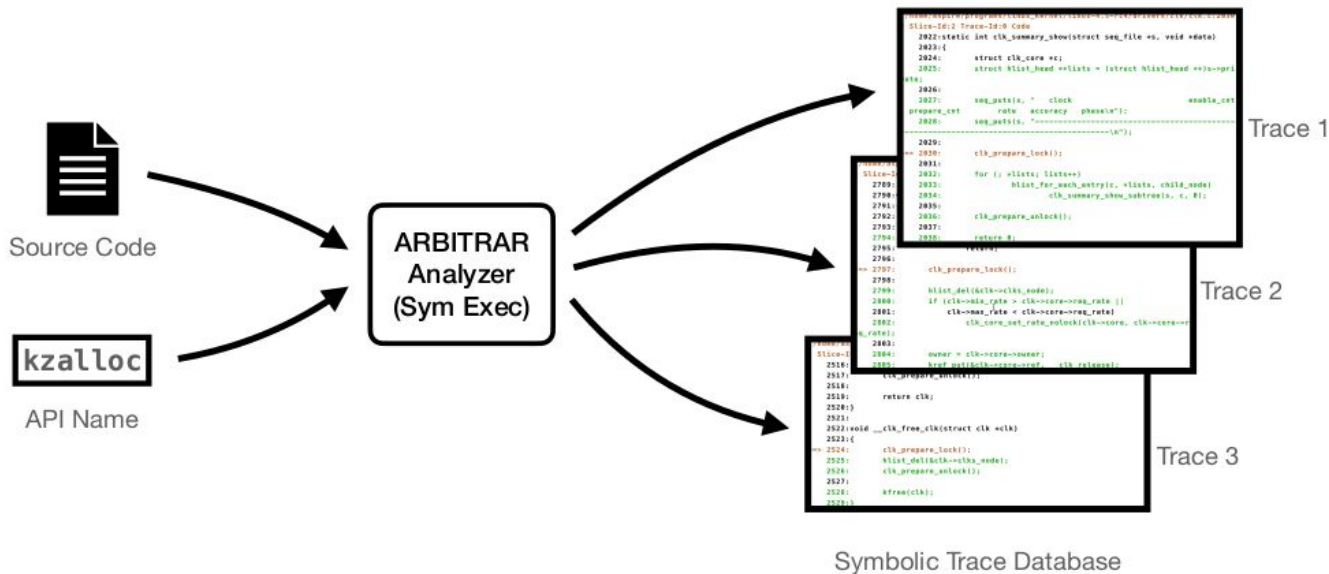
- Can we learn from it?



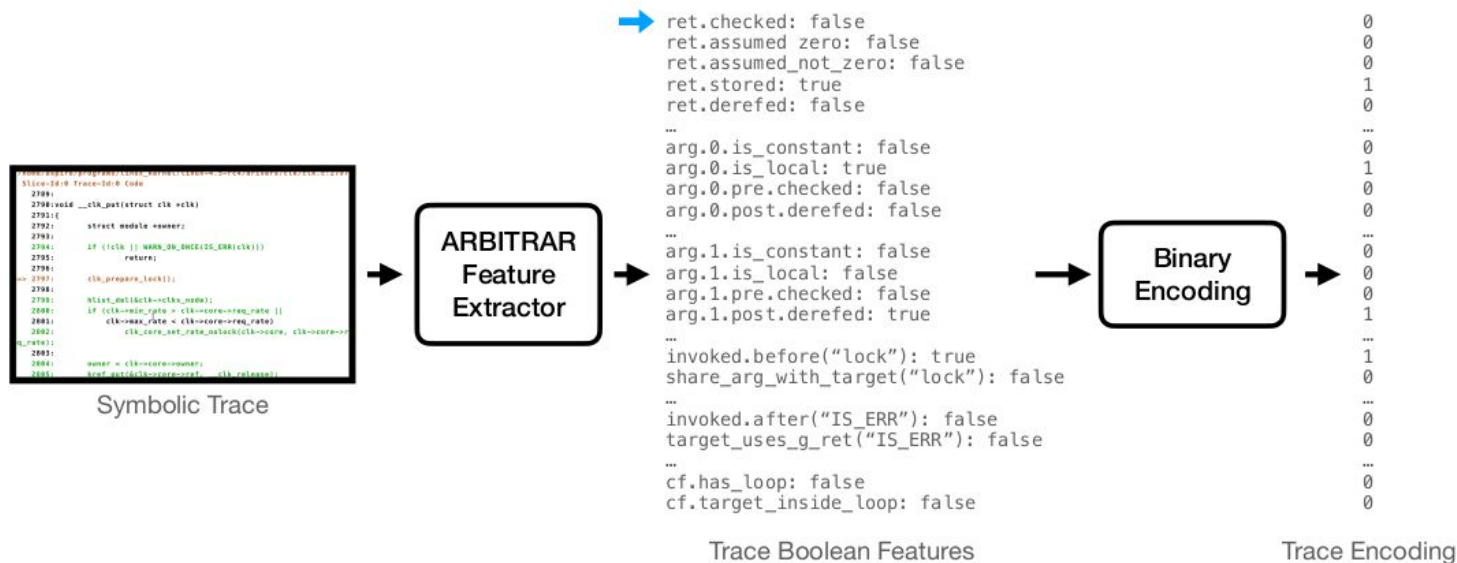
Category 4: ARBITRAR: User-Guided API Misuse Detection!



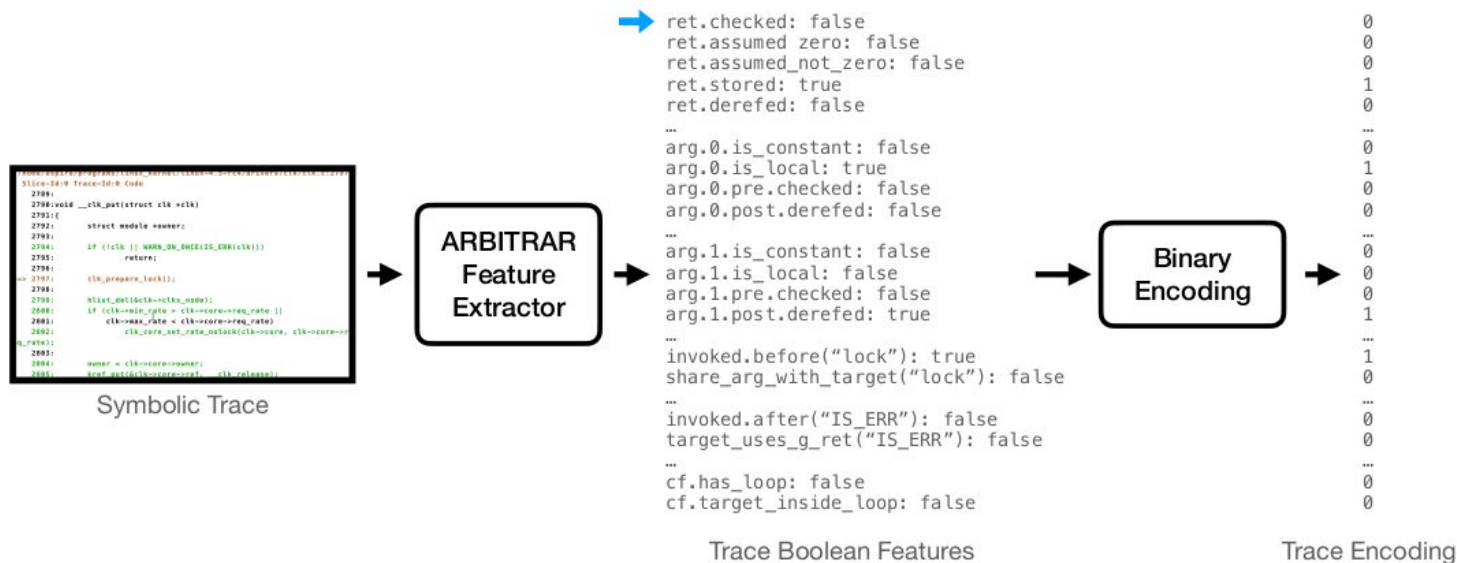
Category 4: ARBITRAR: User-Guided API Misuse Detection!



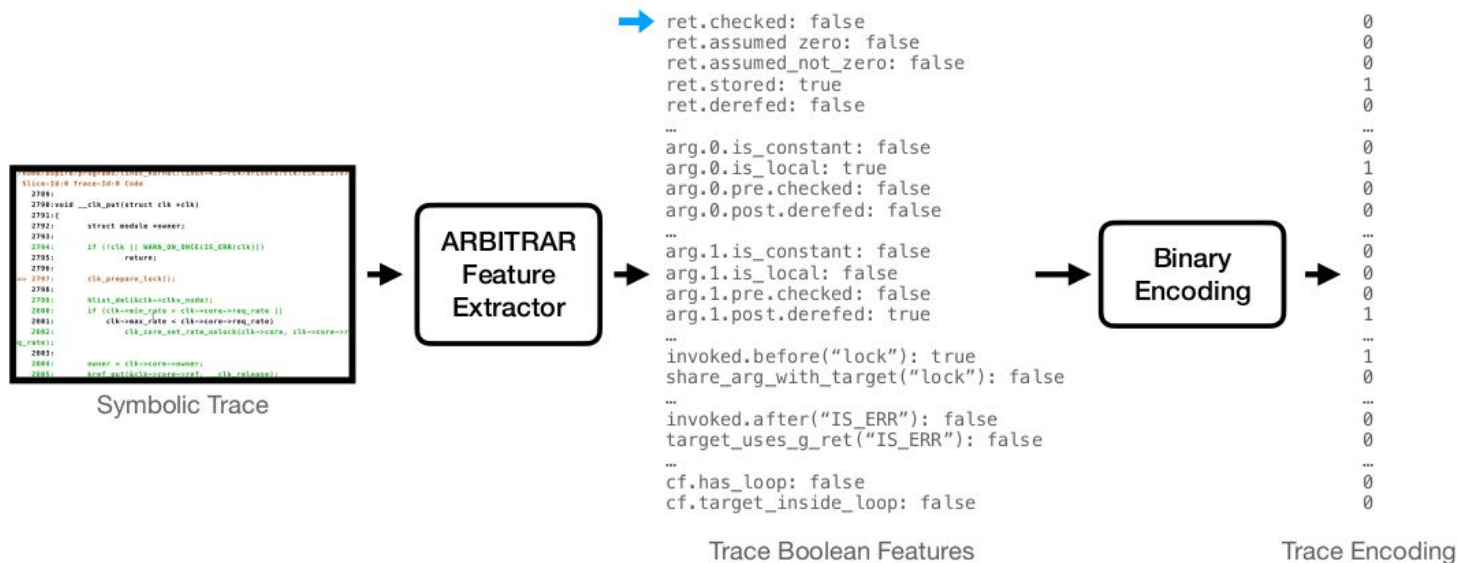
Category 4: ARBITRAR: User-Guided API Misuse Detection!



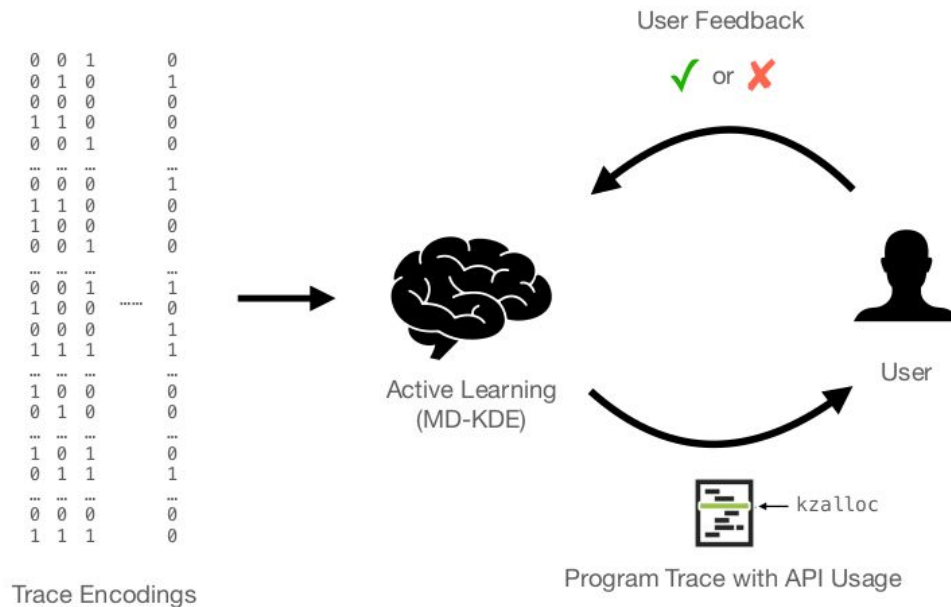
Category 4: ARBITRAR: User-Guided API Misuse Detection!



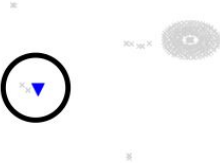

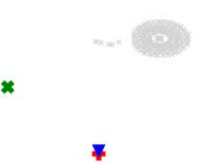

Category 4: ARBITRAR: User-Guided API Misuse Detection!



Category 4: ARBITRAR: User-Guided API Misuse Detection!

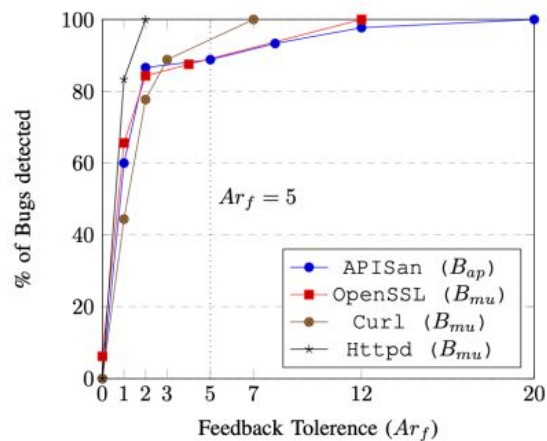


Category 4: ARBITRAR: User-Guided API Misuse Detection!

Iteration	1 st	2 nd	3 rd	4 th
Trace	libpng/png.c:4577 →png_destroy_read_struct(&c.png_ptr, &c.info_ptr, NULL); fclose(fp); return png_error;	pngtools/pngread.c:43 →png_destroy_read_struct(&png_ptr, &info_ptr, ... png_destroy_read_struct(&png_ptr, &info_ptr,	apngasm/apngasm.c:187 →png_destroy_read_struct(&png_ptr, &info_ptr, ... png_destroy_read_struct(&png_ptr, &info_ptr,	apngasm/apngasm.c:195 →png_destroy_read_struct(&png_ptr, &info_ptr,
Info	Correct Usage: As the png_ptr and info_ptr are passed in from arguments	Invalid Usage : Double free because of multiple calls to png_destroy_read_struct	Invalid Usage : Double free because of multiple calls to png_destroy_read_struct	Normal trace
User Feedback	Not a Bug	Bug	Bug	Not a Bug
2D TSNE Plot				
Trace Acquisition	Pick a trace at random.	Pick a trace that is furthest from the previous negative trace.	Pick a trace that is closest to the known positive trace.	Pick a trace that is furthest from the negative trace and closest to the positive trace.

Category 4: ARBITRAR: User-Guided API Misuse Detection!

How quickly did we find bugs? **85% of the bugs were found with first 2 tries.**





Machine Learning Based Methods

- Features depends on bug types:
 - Syntactic v/s semantic features.
- Need large labelled dataset.
- Should have explainable results.
- Interactive techniques should be explored.