SCALABLE AND EFFICIENT PATH-SENSITIVE ANALYSIS TECHNIQUE SCANNING MANY TYPES OF VULNERABILITY

Dongok Kang and Minsik Jin*

PA Division, Fasoo.com R&D Center, Seoul, Repulic of Korea

ABSTRACT

The goal of this paper is to present an efficient and effective path-sensitive analysis technique for many types of security vulnerability. We propose two analysis techniques. The first is a scalable path-sensitive analysis technique for security vulnerability with high precision and recall. Our strategies are to allow flexible design of path state and to make an effective path navigation heuristic which achieves both scalability and high recall. Experimental results show that a vulnerability scanner implemented through this technique get precision 100% and recall 93% on OWASP Benchmark. The vulnerability scanner is able to analyze 1 million lines of code. The second is a pre-analysis technique to improve the efficiency of the above analysis technique. The pre-analysis technique improves the path navigation by using an additional cheap anlysis. Despite the additional cost, experimental results show that the total analysis time is reduced by 2.5 times. Simultaneously recall of the analysis is improved by the pre-analysis technique.

KEYWORDS

Secure coding, Security, Static analysis, Vulnerability scanner, Summary-based, Path-sensitive, Information flow Analysis, Pre-analysis

1. INTRODUCTION

The goal of this paper is to present an efficient and effective path-sensitive taint analysis technique scanning many types of security vulnerability at once. First, we propose an effective path-sensitive analysis technique to achieve high precision and recall. Precision is the ratio of the number of true positives to the number of detected vulnerabilities. Recall is the ratio of the number of true positives to the number of all vulnerabilities. For practical analysis, path-sensitive analysis techniques make use of abstractions of partial paths or symbolic values and path navigation heuristics. Our technique does not care about abstraction to give flexibility to the design of path state. Our technique is easy to achieve high precision because of this feature. It optimizes only path navigation. Therefore path navigation should avoid exploring non-vulnerable paths as much as possible for scalable analysis. On the other hand, path navigation should explore all vulnerable paths as much as possible to achieve high recall. We propose a path navigation

Natarajan Meghanathan et al. (Eds) : ACSIT, SIPM, FCST, CoNeCo, CMIT - 2017
pp. 21– 40, 2017. © CS & IT-CSCP 2017 DOI : 10.5121/csit.2017.70803

strategy satisfying these goals. We evaluate its ability to achieve high precision and recall on OWASP Benchmark. Second, we propose a pre-analysis technique making the path-sensitive analysis several times more efficient. It improves the efficiency of a baseline analysis preserving the original precision and recall. We evaluate the improvement of efficiency on 12 real-world applications. In summary, our goal is

1. Design a scalable path-sensitive analysis technique to scan many types of security vulnerability with high precision and recall.

2. Design a pre-analysis technique to improve the efficiency of the path-sensitive analysis not sacrificing the original precision and recall.

Inter-procedural path-sensitive analysis scanning security vulnerabilities suffers from path explosion problem. The number of paths is exponential to the number of nodes in a control flow graph. Including loop, it may be infinite. All vulnerabilities can be detected by analyzing all the paths, but it is impossible to analyze all of them.

Generally, there are two ways to overcome the path explosion problem in path-sensitive analysis. One way is to analyze each query one by one collecting predicates to be satisfied backward from a query point in a fully path-sensitive manner. It is practical for analyzing small size of queries and has an advantage in precision. A disadvantage is that it is difficult to analyze complex data types and operations. The other way is forward analysis with abstractions of partial paths or values that are irrelevant to interesting properties and with path navigation heuristics selecting paths related to interesting properties.

Forward analysis with path navigation is more viable option for many types of security vulnerability scanner than backward analysis. The size of query points for checking security vulnerabilities is usually large. Just XSS only analysis has too many query points such as JSP's out.write()s. There are several types of security vulnerability such as Command Injection, SQL Injection, Path Traversal, etc. Even if each of them has small size of query points, the sum of them may be large. Backward path-sensitive analysis is not appropriate for scanning many types of security vulnerability because they may have large size of query points. In contrast, forward analysis has an advantage in analyzing many queries at once. We present a path-sensitive analysis technique for security vulnerability that uses forward analysis in Section 3.

However, forward analysis involves a risk to analyze paths that are irrelevant to vulnerabilities. Figure 1 is a motivating example. There are 2 taint sources and 2 sinks. Method SOURCES returns a T1-type tainted value in {1} and a T2-type tainted value in {2}. Method SINKS propagate parameter S to the T1-type sink in {3} and the T3-type sink in {4}. Method PROP has no taint sources or sinks. It propagates a tainted value in parameter X to the P1 in {5} and the P2 in {6}. Method FOO just calls them sequentially: SOURCES, PROP, and then SINKS. A T1-type vulnerability may exist through the path {1}{4}{5}. There are no other vulnerabilities. The most efficient way to scan the vulnerability is making only three path summaries: $(c1, \{\text{return} \mapsto t1\{1\}\}), (\neg c2, \{p2.str \mapsto p2.str + x\}), (c3, \{t1sink\{6\} \mapsto s\})$. By instantiating them, only one path summary is calculated for method FOO: $(c1 \wedge \neg c2 \wedge c3, t1sink\{6\} \mapsto t1\{1\})$. However, there are no inducements to navigate only branch {1} in SOURCES, only branch {4} in PROP, and only branch {5} in SINKS. A path-sensitive analysis has no choice but to result in 8 path summaries for FOO. Despite the time is wasted to analyze non-vulnerable paths, it is difficult to make the path navigation efficient.

We propose a pre-analysis technique for efficient path navigation. The overall process is as follows. A pre-analysis provides program points that may propagate tainted values to their sink points. In Figure 1, it may provide {1}, {4}, and {5}. And then the target path-sensitive analysis utilizes the program points for selecting branches to navigate. The target analysis navigates only branches {1}, {4}, and {5}. In this manner, the prior knowledge provided by the pre-analysis makes aggressive path navigation possible in the target analysis.

A challenging point is that the additional time for the pre-analysis should be less than the reduced time of the target analysis. A poor pre-analysis may give the target analysis invaluable knowledges, for example, all branches in Figure 1. The pre-analysis is not useful to reduce the analysis time of the target analysis. On the other hand, if a pre-analysis spends too much time for valuable knowledge, it is also meaningless work.

```
1  void foo(c1, c2, c3: Bool) {
2    x, y: String
3    p1, p2: StringBuilder
4      = new StringBuilder();
5
6    x = sources(c1);
7    prop(c2, x, p1, p2);
8    sinks(c3, p2.toString);
9  }
10
11 String sources(c1: Bool){
12   if (c1) return t1_src(); //\{1\}
13   else return t2_src(); //\{2\}
14 }
15
16 void prop(c2: Bool, x: String
17   , p1, p2: StringBuiilder){
18   if (c2) p1.append(x); //\{3\}
19   else p2.append(x); //\{4\}
20 }
21
22 void sinks(c3: Bool, s: String) {
23   if (c3) t1_sink(s); //\{5\}
24   else t3_sink(s); //\{6\}
25 }
```

Figure 1: Only exploring {1}, {4}, and {5} is enough to make a summary of method `foo`

The pre-analysis technique involves a risk that a selected branch may block exploring another selected branch. Pre-analysis considering only propagations of tainted values do not give a knowledge of feasible paths. Not only points related to tainted values but also points related to calculation of branch conditions should be considered for path navigation. Figure 2 shows an example of the problem. ZIPFILE contains a tainted value and the sink is in PROCESSFILE of line 9. It is impossible to execute code at line 9 without passing code at line 6. Method PROCESSPROPERTYFILE contains two branches and each of them returns different boolean

value. For exploring line 9, PROCESSED at line 8 should have false value. Therefore a path navigation should explore the false branch in PROCESSPROPERTYFILE.

We solve the problem by marking may-feasible partial paths in advance. Using heuristic methods, the pre-analysis finds intervals that contain pairs of a definition of a variable and a branching condition using the variable. We call the intervals as effective intervals. In Figure 2, ([line 3], [line 9]) is an effective interval. The pre-analysis calculates infeasible partial paths in the intervals by path-sensitive constant propagation. And then the other paths in the intervals are marked as may-feasible paths. In Figure 2, [line 3][line 6][processPropertyFile; line 22][line 9] is a may-feasible path in the effective interval ([line 3], [line 9]). The target analysis utilizes the knowledge to avoid infeasible paths. We discuss the reason why this strategy, constant propagation within small code interval, effectively solves the problem in Section 2.

This paper is organized as follows. Section 2 introduces previous researches about path-sensitive analysis. Section 3 introduces a scalable and effective path-sensitive analysis technique. Section 4 explains about a pre-analysis technique to make the path-sensitive analysis more efficient. Section 5 explains experimental results and discuss about pros and cons of our analysis techniques.

```
1  public Plugin extractJarFile(...){
2    ...
3    boolean processed =
4      processClassFile(zipEntry);
5    if (!processed) {
6      processed=processPropertyFile(...);
7    }
8    if (!processed) {
9      processFile(plugin, zipFile,
10       zipEntry, targetDirectory);
11       ...
12 }
13
14 private boolean processPropertyFile(...)
15     throws IOException {
16   if (zipEntry.getName().
17       endsWith(".properties")) {
18     final File targetFile=new File(...);
19     copyFile(...);
20     return true;
21   }
22   return false;
23 }
```

Figure 2: An example code from PluginExtractor.java of Webgoat. Path navigation of line 9 is required for detecting a vulnerability. It is possible only when the analysis explore the false branch of `processPropertyFile`.

2. RELATED WORK

In this section we survey previous approaches to path-sensitive analysis. Path-sensitive analysis technique was used for various purposes such as points-to analysis, security vulnerability scanner, memory leak detection, compiler optimization, and language refinement. They fall into two categories: backward analysis and forward analysis. Backward analysis techniques are used for one special property. Each analysis technique includes optimization methods to overcome path explosion problem. The optimizations are performed by ignoring or abstracting paths and values unrelated to the properties of interests.

[9] avoids unnecessarily explored path by join that do not hurt accuracy. The work is to find infeasible paths for compiler optimization. They formalize edge strings and discuss optimizations. They explain a concept of delayed join. The method uses k-edge abstraction inspired by k-context abstraction. They insist 2-sensitive edge strings are enough for finding infeasible paths. This fact inspired us to use a heuristic using small size of codes for finding infeasible paths. Although it is difficult to make use of this work as it is, it shows that there are localities between definitions of a variable and the uses of the variables in branch conditions.

[11] presents an optimization technique abstracting away certain symbolic subterms to make the analysis practical. By abstracting elements unrelated to a property of interest, it reduces analysis time. They test only small codes to show the efficiency.

[8] presents a vulnerability detection technique using a backward analysis. Program condition(PC) and security condition(SC) are constructed to prove a safety of a query point. It is proved by solving $\mathrm{PC} \wedge \overline{SC}$. The weak point of this work is that it does not handle complex data types because of the limitation of backward analysis.

[7] presents a demand-driven analysis technique for buffer overflow detector. The technique is backward analysis for buffer access violation. It is the efficient way to analyze the property. The method makes use of another path-insensitive pointer analysis. The requirement of time-consuming pointer analysis is a limitation of this work.

[1] solves infeasible paths problem by abstract interpretation. The method refines syntactic language for another path-sensitive analysis. It gives us a lesson that removal of infeasible path is important to prove more queries. It only concentrates on improvement of true positive rate. The cost of infeasible-path detection increases the total analysis time.

[3] presents a technique merging branches unrelated to properties of interest. It gives us a lesson that avoiding unrelated branch is important to make analysis efficient. Proposed method is difficult to be used as it is for making vulnerability scanner because a vulnerability scanner has to consider sanitizer.

[15] presents a technique to detect RCE vulnerability. The method slices codes related to sinks and checks the satisfiability of a sink's collected predicates.

[12] uses a global invariant approach to detect data race avoiding the path explosion problem. They do not present an experimental result that shows scalability to real-world programs.

[5] shows a method to add path sensitivity to points-to analysis. Their approach works with both WPP2G and P2SSA representations yielding a certain degree of path-sensitivity. It is a technique abstracting elements unrelated to the property of interest.

[6] presents a backward slicing technique that excludes spurious dependencies lying on infeasible paths and avoids imprecise join. It traverse a program in a depth-first search manner and reuse dependencies from precomputed paths. The novelty of this work is a formalization of efficiently reusing conditions. Experimental results of the prototype does not show scalability.

[14] presents a scalable path-sensitive analysis for memory leak detection. It is a great work that concentrates on memory leak detection. They presents efficient summarizing method for summaries to get scalability.

[13] presents an optimized backward analysis technique for data flow analysis. They optimize the path-sensitive analysis by abstract subterms as unknown values in a predicate. They present a calculation of predicates containing unknown terms.

[4] presents a formal method for computing the precise necessary and sufficient conditions for program properties that are fully context- and path-sensitive. They show a sound, complete and scalable analysis for only small functional language. Experimental results of the prototype on real-world programs does not show high precision.

3. PATH-SENSITIVE ANALYSIS FOR SECURITY VULNERABILITY

The target program is a list of bodies composed of a list of statements and a control flow graph. Figure 3 shows the target program. pgm means a program. It is list of bodies with method name $((\text{body}_f)+)$. A body consists of a list of statements and a control flow graph. The statements consist of assignment statement $(x := e)$, assert statement $(\text{assert}(x \odot e))$, and call statement $(\text{call}(f, x))$. Expression e is composed of typical expressions such as arithmetic expressions or logical expressions, l-value lv, and primitive values primitive.

$$
\begin{array}{rcl}
pgm & \rightarrow & (body_f)+ \\
body & \rightarrow & ((stmt_{node})+, g) \\
stmt & \rightarrow & x := e \\
 & | & \texttt{assert}(x \odot e) \\
 & | & \texttt{call}(f, x) \\
e & \rightarrow & e \oplus e \mid lv \mid primitive \\
lv & \rightarrow & lv.\text{field} \mid x \\
g & \rightarrow & \phi \\
 & | & g \cup \{node \hookrightarrow node\} \\
node & \rightarrow & \texttt{Entry} \mid \texttt{Exit} \mid stmt\_index
\end{array}
$$

Figure 3: A program contains statements and control flow graphs

Our path-sensitive analysis technique uses summary-based analysis technique. A summary-based analysis analyzes each method in bottom-up call order generating a summary consists of symbolic states. Summaries are instantiated in each call statement making use of each caller's context. Summary-based analysis is a typical choice for path-sensitive analysis because of scalability. Figure 4 shows the summary design of the analysis. Summary describes the

summary. A summary is a set of path summaries. A path summary is a symbolic state of a path. A path summary consists of a path string Path, last node's symbolic memory SMemory, and each symbolic value's constraints SConstraint. We do not present the specific designs of SConstraint and SMemory because our path-sensitive analysis technique does not care about the complexity of the summary design. Our goal is to get scalability not by a low-cost summary but by a path navigation strategy.

$$\begin{aligned} \mathit{Summary} &= \wp(\mathit{PathSummary}) \\ \mathit{PathSummary} &= \mathit{Path} \times \mathit{SConstraint} \times \mathit{SMemory} \\ \mathit{Path} &= (\mathit{Method} \times \mathit{Node})\text{List} \\ \mathit{SConstraint} &= \mathit{SVal} \to \mathit{Constraint} \\ \mathit{SMemory} &= \mathit{SRef} \to \mathit{SVal} \end{aligned}$$

Figure 4: Summary is a set of path states

Figure 5 shows the semantics of our analysis technique. It calculates output path summaries from an input path-summary. next(f, i) is successor nodes of a node (f, i) in the control flow graph. An assignment statement changes the state of symbolic memory by storing evaluated symbolic value $[[e]]m$ to the evaluated location $[[x]]_{lv}m$. An assert statement changes the state of symbolic constraint by storing the evaluated constraint. If an evaluated constraint is proved to be false, the path summary is discarded. The analysis propagates summaries of each method bottom-up through the call order. At a call statement, it lookups the callee's precomputed summaries $SUM(f')$ and instantiate it in the caller's context. In this way, it inter-procedurally detects vulnerable paths.

We introduce the details of our path sensitive analysis technique. The analysis explores local path in a depth-first search manner and filters most likely to be vulnerable paths among several inter-procedural paths. Algorithm 1 describes the details. It calculates path summaries visiting local nodes. WEIGHTPATHFILTER is a heuristic algorithm for filtering path summaries that are most likely to be vulnerable. It is applied to the intermediate path summaries for each iteration. At a call statement of a local path, there may be n inter-procedural paths with each symbolic state and m path summaries. After instantiation, there are, at most, $n \times m$ inter-procedural paths with each symbolic state. At this point, WEIGHTPATHFILTER reduces the size of path summaries. The path navigation halts when 100% of node coverage is achieved or iteration is over the threshold.

$$\begin{aligned} &[\![\cdot]\!]\cdot : \mathit{Node} \to \mathit{PathSummary} \to \wp(\mathit{PathSummary}) \\ &[\![(x := e)_i]\!](p,c,m) \\ &= \left\{ (p',c,m') \;\middle|\; \begin{array}{c} (f,j) \in \mathsf{next}(f,i) \\ p' = p :: (f,j) \\ m' = m\{[\![x]\!]_{\mathsf{lv}} m \mapsto [\![e]\!]m\} \end{array} \right\} \\ &[\![\mathtt{assert}(x \odot e)_i]\!](p,c,m) \\ &= \left\{ (p',c',m) \;\middle|\; \begin{array}{c} (f,j) \in \mathsf{next}(f,i) \\ p' = p :: (f,j) \\ \mathit{constr} = [\![x \odot e]\!]_c m \neq \text{false} \\ c' = c\{[\![x]\!]m \mapsto \mathit{constr}\} \end{array} \right\} \\ &[\![\mathtt{call}(f',x)_i]\!](p,c,m) \\ &= \left\{ (p',c',m') \;\middle|\; \begin{array}{c} (f,j) \in \mathsf{next}(f,i) \\ (p_{f'},c_{f'},m_{f'}) \in \mathsf{SUM}(f') \\ p' = p :: p_{f'} :: (f,j) \\ m' = \mathsf{INST}_{\mathsf{m}}(m_{f'},m,x) \\ c' = \mathsf{INST}_{\mathsf{c}}(c_{f'},m,x) \neq \text{false} \end{array} \right\} \end{aligned}$$

Figure 5: Analysis semantics

Algorithm 1 Summary Calculation with Path Navigation Heuristic

```
1: procedure CALCUATESUMMARY(f, body)                          ▷ The summary of method f
2:     result_sum ← ϕ, sum ← {((f, Entry), ϕ, ϕ)}
3:     cnt ← 0, visit ← ϕ
4:     while sum ≠ ϕ ∧ unvisited nodes exist ∧ iter < threshold do
5:         sum_pop, sum ← pop_one_local_path(sum)              ▷ Depth-first Traversal
6:         visit ← sum_pop's last node
7:         iter ← iter + 1
8:         sum' ← {s' | s ∈ sum_pop, s' ∈ [[last(s.p)]](s)}    ▷ Evaluate semantics
9:         sum ← sum ∪ WEIGHTPATHFILTER(sum')                  ▷ Select top-ranked paths through WPF
10:        terminated, sum ← pop_terminated(sum)
11:        result_sum ← result_sum ∪ terminated
12:    end while
13:    result_sum ← WEIGHTPATHFILTER(result_sum)               ▷ Select top-ranked result paths
14:    return result_sum                                       ▷ The summary of method f is result_sum
15: end procedure
```

WEIGHTPATHFILTER algorithm is described on Algorithm 2. It simply sorts path summaries according to their respective scores and picks the top-ranked path summaries. The sorting criteria W contains several heuristics. Mainly, path summaries with following properties are preferred.

- many tainted values and sinks.
- many parameter symbols.
- short path.
- many equal constraints.

Path summaries having more parameter symbols are preferred than others because it has higher possibility of propagating tainted values. Path summaries having shorter paths are preferred than others because most of vulnerabilities have short vulnerable paths. Equal constraints are useful for pruning infeasible paths, therefore it is preferred.

Algorithm 2 Heuristic: Weight-guided Path Filter

```
1: procedure WEIGHTPATHFILTER(sum : Summary)
2:     weight_sorted ← sort(sum, W)                            ▷ W : Summary → Score
3:     result ← filter_high_weight(weight_sorted, Threshold)
4:     return result
5: end procedure
```

The introduced path-sensitive analysis technique is scalable and has an ability to achieve high precision and recall. We evaluate the performance in Section 5.

4. PRE-ANALYSIS TECHNIQUE

We present a pre-analysis technique to improve efficiency of a target path-sensitive analysis. In Section 1, we address an inefficiency of path-sensitive analysis which is difficult to be improved. To avoid path navigation of non-vulnerable paths, we propose a pre-analysis technique providing

prior-knowledge for target path-sensitive analysis. We call the prior knowledge as Path-oracle, the pre-analysis as Pre-analysis, and the target path-sensitive analysis as Main-analysis.

4.1. Path-oracle

Here, we describe Path-oracle provided to Main-analysis by Pre-analysis. Figure 6 shows the Path-oracle description. Path-oracle consists of three part: Pick, San, and Allowed. Pick is a set of nodes that may be included in vulnerable paths. San is a set of nodes that may sanitize tainted values. Allowed is a set of may-feasible partial paths. The form of Allowed is a map from a starting node to set of may-feasible partial paths. Allowed is used to avoid infeasible paths by reserving the paths in each path summary

$$
\begin{aligned}
O \in PathOracle &= InfoFlowHint \\
&\quad \times SanitizingHint \\
&\quad \times AllowedPaths \\
Pick \in InfoFlowHint &= Node \to Bool \\
San \in SanitizingHint &= Node \to Bool \\
Allowed \in AllowedPaths &= Node \to \\
&\quad \wp(ReservedPath)
\end{aligned}
$$

Figure 6: Path Oracle

4.2. Challenge of Pre-analysis

A Pre-analysis should achieve the following goals to be useful.

- Lightweight analysis
- Conservative detection of vulnerabilities
- Accurate enough to provide useful Path-oracle.

A Pre-analysis should occupy little portion of total analysis time. Our goal of the pre-analysis technique is reducing the total analysis time by improving the efficiency of a Main-analysis. If the analysis time of a Pre-analysis is longer than the reduced time of the Main-analysis, it is useless.

A Main-analysis makes use of a Path-oracle aggressively to its path navigation, therefore it should detect all vulnerable paths as possible. A Main-analysis tries to visit only nodes in Pick. Therefore, the result of a Pre-analysis should include all possible nodes included in vulnerable paths.

Too imprecise Pre-analysis result misleads the Main-analysis into wasting time. A Main-analysis tries to cover all nodes in Pick. Unnecessarily selected nodes in a Pre-analysis lead to unnecessary path navigations in the Main-analysis. A Pre-analysis should provide essential nodes only.

4.3. Pre-analysis Memory Model

A Pre-analysis makes use of abstract interpretation[2]. A Pre-analysis is context/flow-insensitive and field-insensitive. It has only one memory state $d \in D$ for all program points. Figure 7 shows the design. The abstract location of memory state is variable. Each field strings are ignored.

A Pre-analysis collects vulnerable variables and nodes. Source map ∈ Src and sink map ∈ Sink consist of a map from variable to related taint rules and a map from node to related taint rules. The variable map is for analyzing flow of taint rules. The node map is for producing Path-oracle. A Pre-analysis collects only nodes for sanitizer, because it is not interested in sanitizing flows.

Domain	$d \in \mathbb{D}$	$=$	$\{\texttt{taint} : \mathbb{T}, \texttt{alias} : \mathbb{A}\}$
Taint map	$\mathbb{T}$	$=$	$\{\texttt{src} : Src, \texttt{sink} : Sink$, $\texttt{san} : San\}$
Source map	Src	$=$	$(Var \to TRules) \times (Node \to TRules)$
Sink map	$Sink$	$=$	$(Var \to TRules) \times (Node \to TRules)$
Sanitizer map	San	$=$	$Node \to TRules$
Taint rule set	$TRules$	$=$	$\wp(TRule)$
Alias relation	$\mathbb{A}$	$=$	$Var \to \wp(Var)$

Figure 7: Memory model of Pre-analysis

4.4. Pre-analysis Semantics

Figure 8 shows the semantics on the memory model described in Figure 7. It describes transition rules of $\mathbb{D}$ for each statement. Notice that it does backward propagation for detecting the flows to a sink. And it does forward propagation for detecting the flows from a source. It collects alias information to detect taint propagation by heap location.

Figure 9 describes field-insensitive semantics about l-value evaluation. $[\![\text{lv.f}]\!]_{\text{lv}}\text{d} = [\![\text{lv}]\!]_{\text{lv}}\text{d}$ presents that we do not care about field. $[\![\text{x}]\!]_{\text{lv}}\text{d} = \{\text{x}\} \cup \text{d.alias(x)}$ presents that it considers alias while evaluating l-value.

4.5. Pre-analysis Algorithm

A Pre-analysis calculates approximate fixed point of D. In Algorithm 3, codes at line 5-10 calculate it. During the iteration, taint rules of sources and sinks are propagated through the semantics. For efficiency, the while loop halts when its iteration exceeds threshold. It halts when $\mathbb{d}$ is saturated through the semantics.

A Pick is made with Src and Sink of $\mathbb{D}$. A Pre-analysis intersects each node's taint rules in Src and Sink to find vulnerable nodes. The initial Pick is made of the intersected taint rules. Code at line 13 in Algorithm 3 then propagates each taint rules of the nodes in the Pick to its dominator nodes. This process is described in Figure 10 (a), (b). Before line 13, it looks like (a). Propagation nodes of sources and sinks are only picked. After line 13, it looks like (b). A path related to the vulnerability is constructed naturally.

$$
\begin{array}{rcl}
[\![\cdot]\!]_{\mathsf{pre}\cdot} & : & \mathit{Stmt} \to \mathbb{D} \to \mathbb{D} \\
\\
[\![lv := e]\!]_{\mathsf{pre}} d & = & \\
\left\{ \begin{array}{l}
d.\mathtt{taint.src} := \\
d.\mathtt{taint.src}\{x \mapsto d.\mathtt{taint.src}(x) \cup [\![e]\!]d \mid x \in [\![lv]\!]_{\mathsf{lv}} d\} \\
\qquad \{\mathit{node} \mapsto d.\mathtt{taint.src}(\mathit{node}) \cup [\![e]\!]d\} \\
d.\mathtt{taint.sink} := \\
d.\mathtt{taint.sink} \\
\qquad \{y \mapsto d.\mathtt{taint.sink}(y) \cup [\![lv]\!]d \mid y \in [\![e]\!]_{\mathsf{lv}} d\} \\
\qquad \{\mathit{node} \mapsto d.\mathtt{taint.sink}(\mathit{node}) \cup [\![lv]\!]d\} \\
d.\mathtt{alias} := \\
d.\mathtt{alias}\{x \mapsto X \cup \{y\},\ y \mapsto Y \cup \{x\} \mid \\
\qquad x \in X = [\![lv]\!]_{\mathsf{lv}} d,\ y \in Y = [\![e]\!]_{\mathsf{lv}} d\}
\end{array} \right. \\
[\![\mathtt{assert}(x \odot e)]\!]_{\mathsf{pre}} d & = & d \\
[\![r := \mathtt{call}(f, x)]\!]_{\mathsf{pre}} d & = & \left[\!\!\left[\begin{array}{c} x := f.\mathsf{param} \\ f.\mathsf{param} := x \\ r := f.\mathsf{ret} \end{array} \right]\!\!\right]_{\mathsf{pre}} d \\
& & (\text{inter-procedural propagation}) \\
[\![\mathtt{source}(x, T)]\!]_{\mathsf{pre}} d & = & \\
\qquad d.\mathtt{taint.src}\{x \mapsto d.\mathtt{taint.src}(x) \cup \mathsf{SRC}(T)\} & & \\
[\![\mathtt{sink}(y, T)]\!]_{\mathsf{pre}} d & = & \\
\qquad d.\mathtt{taint.sink}\{y \mapsto d.\mathtt{taint.sink}(y) \cup \mathsf{SINK}(T)\} & & \\
[\![\mathtt{san}(x, T)]\!]_{\mathsf{pre}} d & = & d.\mathtt{taint.san}\{\mathit{node} \mapsto \mathsf{SAN}(T)\}
\end{array}
$$

Figure 8: Semantics of Pre-analysis

$$
\begin{array}{rcl}
[\![\cdot]\!]_{\mathsf{lv}\cdot} & : & \mathit{Expr} \to \mathbb{D} \to \wp(\mathit{Var}) \\
[\![e \oplus e]\!]_{\mathsf{lv}} d & = & [\![e]\!]_{\mathsf{lv}} d \cup [\![e]\!]_{\mathsf{lv}} d \\
[\![lv.f]\!]_{\mathsf{lv}} d & = & [\![lv]\!]_{\mathsf{lv}} d \\
[\![x]\!]_{\mathsf{lv}} d & = & \{x\} \cup d.\mathtt{alias}(x)
\end{array}
$$

Figure 9: Evaluation rule for I-value in Pre-analysis

A San is made with San of D. Different from Pick, it only propagates each nodes to their mutually dominant nodes. A Main-analysis uses San passively in path navigation. This is why a Pre-analysis propagates sanitizing nodes passively. In Figure 10 (c), the square node is sanitizing node. After line 16, it looks like (d). It changes only node {3} as a sanitizing node. Since the Main-analysis knows {3} is an entrance to the sanitizer, it may explore a vulnerable path {1}{2}{5}{6}.

An Allowed is made with constant propagation on effective_interval. In line 18 of Algorithm 3, scan_effective is an effective interval classifier. For example, see Figure 10 (e). Interval ({2}, {9}) is an effective interval. Last node {9} is picked as propagation node. There are many possible paths between {2} and {9} and two boolean value assignments exist. A Pre-analysis does path-sensitive constant propagation on this interval. The Pre-analysis may discover two path {2}{3}{7}{9}, {2}{4}{6}{7}{9} are infeasible. For finding effective interval, it uses several heuristics such as the number of boolean assignment. Low cost for finding those effective intervals is important.

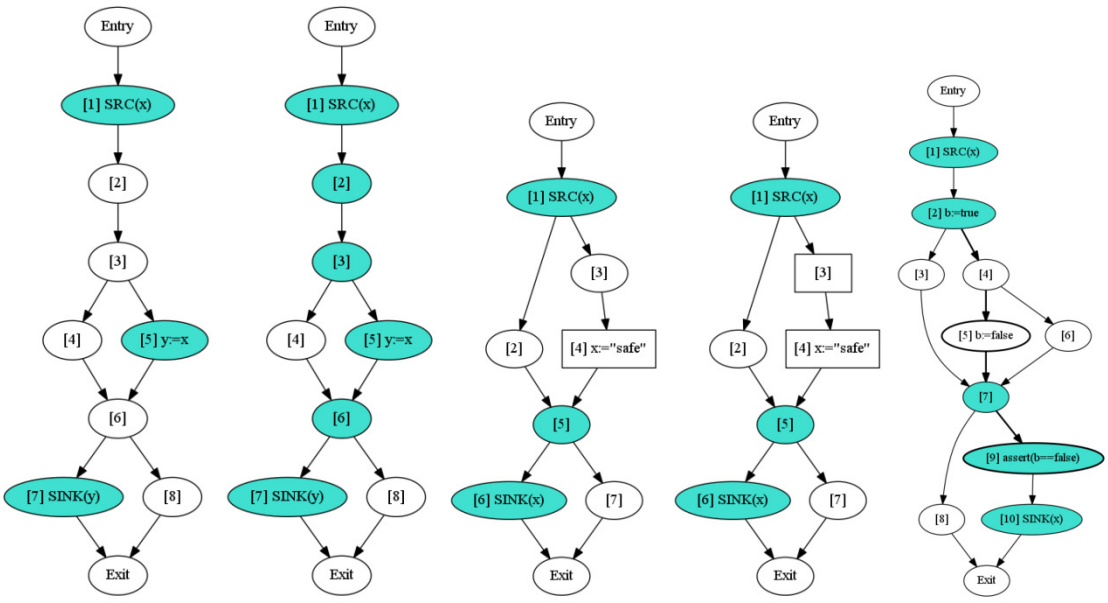
Algorithm 3 Pre-analysis: Path-oracle Calculation

procedure CALCULATEPATHORACLE(pgm)
  $d \leftarrow \{\texttt{taint} : \{\texttt{src} : \phi, \texttt{sink} : \phi, \texttt{san} : \phi\}, \texttt{alias} : \phi\}$
  $saturated \leftarrow false$
  $iter \leftarrow 0$
  while $\neg saturated \vee iter < threshold$ **do** ▷ Calculate least fixed point of semantics $[\![\cdot]\!]_{\mathsf{pre}}$.
    $iter \leftarrow iter + 1$
    $d' \leftarrow d$
    $\forall body_f \in pgm.\ \forall stmt_i \in body_f.\ d \leftarrow [\![stmt_i]\!]_{\mathsf{pre}} d'$
    $saturated \leftarrow d' = d$
  end while
  $default \leftarrow \{node \mapsto \text{false} \mid node \in AllNode\}$
  $Pick \leftarrow default\{node \mapsto \text{true} \mid (d.\texttt{src}(node) \cap d.\texttt{sink}(node)) \neq \phi\}$ ▷ Both have info-flow relationship
  $Pick \leftarrow Pick\{d \mapsto \text{true} \mid d \text{ dominates } n,\ Pick(n) = true\}$ ▷ To find a way to the info-flow-important node

  $San \leftarrow default\{node \mapsto \text{true} \mid d.\texttt{san}(node) \neq \phi\}$
  $San \leftarrow San\{d \mapsto \text{true} \mid d \text{ dominates } n,\ n \text{ postdominates } d,\ San(n) = true\}$

  $effective\_interval \leftarrow \mathsf{scan\_effective}(pgm, Pick)$ ▷ [*start_node*,picked *end_node*]
  $Allowed \leftarrow \phi$
  for each $(node::path_{\mathsf{rsvd}})$ **in** $\mathsf{paths}(effective\_interval)$ **do**
    $Allowed \leftarrow \{node \mapsto (Allowed(node) \cup \{path_{\mathsf{rsvd}}\}) \mid (node :: path_{\mathsf{rsvd}}) \text{ is not proved to be infeasible}\}$
  end for

  Return $(Pick, San, Allowed)$ ▷ Return the Path-oracle O
end procedure



(a) Initial Pick (b) Propagated Pick (c) Initial San (d) Propagated San (e) Allowed

Figure 10: The process of calculating Path-oracle.

4.6. Path-oracle Guided Path Navigation

Algorithm 4 shows an algorithm filtering path summaries with Path-oracle. At first, it filters infeasible paths proved by Allowed out. And then, if current nodes of summary are assert statements, it selects the branches to continue searching using Path-oracle.

Codes at 5-12 lines filter infeasible path out by reserving only paths in Allowed. An extended summary design including reserved path is described in Section 4.7. Code at line 8 in Algorithm 4 reserves paths in Allowed for each path summaries. In Figure 10(e), a path {4}{5}{7}{9} is reserved. Naturally, unreserved paths, that is, infeasible paths will be excluded in the future summary calculation.

Codes at 14-30 lines seek effective next node using Pick and San. At first, if the branch condition's truth value is determined, it selects the branch having true assert condition. In the other case, it selects branches whose Pick is true or opposite branch has true San. In Figure 10 (b). branch {5} is selected because {5} is in Pick. In (d), branch {2} is selected because {3} is in San. In case neither Pick nor San exist, it selects one branch arbitrarily avoiding already pruned branch. Because both branches are not affective to vulnerable behaviour.

Algorithm 4 Path-oracle-guided Path Filter

```
1:  procedure ORACLEPATHFILTER(sum : Summary, O: PathOracle)
2:      (Pick : Node → Bool, San : Node → Bool, Allowed : Node → ℘(ReservedPath)) ← O
3:      result ← ϕ
4:      nodes ← last nodes of sum
5:      for each s in sum do
6:          if last(s.p) ∈ DOM(Allowed) then
7:              reserved_path_set ← Allowed(last(s.p))
8:              result ← result ∪ {(r, s.p, s.c, s.m) | r ∈ reserved_path_set}
9:          else
10:             result ← result ∪ {s}
11:         end if
12:     end for
13:
14:     if nodes = {j1, j2} ∧ j1 is assert(e), j2 is assert(¬e) then
15:         j1sum ← {s | s ∈ result, last(s.p) = j1}
16:         j2sum ← {s | s ∈ result, last(s.p) = j2}
17:         if [[e]]c m = true then result ← j1sum
18:         else if [[¬e]]c m = true then result ← j2sum
19:         else                                          ▷ [[e]]c m = don't know
20:             if (Pick(j1) ∨ San(j2)) ∧ ¬(Pick(j2) ∨ San(j1)) then
21:                 result ← j1sum                        ▷ j1 branch is only picked
22:             else if ¬(Pick(j1) ∨ San(j2)) ∧ (Pick(j2) ∨ San(j1)) then
23:                 result ← j2sum                        ▷ j2 branch is only picked
24:             else if ¬(Pick(j1) ∨ San(j2)) ∧ ¬(Pick(j2) ∨ San(j1)) then
25:                 if j1sum ≠ ϕ then result ← j1sum      ▷ Arbitrarily j1 picked
26:                 else result ← j2sum                   ▷ Arbitrarily j2 picked
27:                 end if
28:             end if
29:         end if
30:     end if
31:     return result                                     ▷ Finally selected path summaries
32: end procedure
```

Figure 11 is an example of San is essential for path navigation. In code of line 7, GETREQUESTEDSESSIONID gives source information and then it goes to sink at line 11. There are two branches between line 7, 11. Both two branches are not nodes propagating the source information to the sink. However, true branch is actually safe branch. Path navigation should avoid sanitizer to detect the vulnerable path.

4.7. Pre-analysis Technique Guided Path-sensitive Analysis for Security Vulnerability

We present A Main-analysis using Path-oracle for its path navigation. Figure 12 shows a new Summary design. There are only minor changes. ReservedPath field is added in PathSummary. A path summary having reserved path should be calculated following the reserved path. It is shown in the Figure 13.

```
1  void doGet(HttpServletRequest req
2      , HttpServletResponse resp) throws
3        ServletException, IOException {
4    resp.setContentType("text/plain");
5    PrintWriter pw = resp.getWriter();
6    String sessionId =
7      req.getRequestedSessionId(); //SRC
8    if (sessionId == null) {
9      sessionId = "none";//Sanitizing
10   }
11   pw.write(sessionId); //SINK
```

Figure 11: Example of code including a sanitizing branch from TestCoyoteAdapter.java of Tomcat

$$
\begin{array}{rcl}
\mathit{Summary} & = & \wp(\mathit{PathSummary}) \\
\mathit{PathSummary} & = & \{\mathtt{r} : \mathit{ReservedPath}, \mathtt{p} : \mathit{Path} \\
 & & , \mathtt{c} : \mathit{SConstraint}, \mathtt{m} : \mathit{SMemory}\} \\
\mathit{ReservedPath} & = & (\mathit{Method} \times \mathit{Node})\mathrm{List} \\
\mathit{Path} & = & (\mathit{Method} \times \mathit{Node})\mathrm{List} \\
\mathit{SConstraint} & = & \mathit{SVal} \rightarrow \mathit{Constraint} \\
\mathit{SMemory} & = & \mathit{SRef} \rightarrow \mathit{SVal}
\end{array}
$$

Figure 12: Summary of Main-analysis

$$
\begin{array}{l}
[\![stmt_i]\!]_{\mathsf{new}}(\phi, p, c, m) \\
= \{(\phi, p', c', m') \mid (p', c', m') \in [\![stmt_i]\!](p, c, m)\} \\
[\![stmt_i]\!]_{\mathsf{new}}(p_{\mathsf{rsvd}} :: r, p, c, m) \\
= \left\{ (r, p', c', m') \;\middle|\; \begin{array}{c} (p', c', m') \in [\![stmt_i]\!](p, c, m) \\ p' = p :: p_{\mathsf{rsvd}} \end{array} \right\}
\end{array}
$$

Figure 13: Semantics of Main-analysis

Algorithm 5 is extended algorithm of Algorithm 2 by the new semantics and ORACLE PATHFILTER algorithm. In line 8, the previous semantics is replaced by the new semantics. At line 9 ORACLEPATHFILTER algorithm is added. It reserves may-feasible paths for some path summaries and filters out non-vulnerable path summaries. WEIGHTPATHFILTER is preserved same as before to ensure the quality of the previous.

5. EXPERIMENT RESULT

We evaluate the proposed two analysis techniques by two experiments. First, we evaluate their ability to achieve high precision and recall on OWASP Benchmark. OWASP Benchmark is a benchmark to evaluate precision and recall of a vulnerability scanner. The result is presented in Section 5.1. Second, we evaluate the improvement by the pre-analysis technique on 12 real-world projects.

We implemented two analyzer for the experiments. One analyzer is implemented through the path-sensitive analysis technique. The other analyzer is implemented through the pre-analysis technique based on the first analyzer. We call the first one as Baseline analyzer and second one as Modified analyser. t be published in the conference proceedings.

5.1. Experiment 1: Precision and Recall

Algorithm 5 Main-analysis: Summary Calculation with Path-oracle-guided Path Navigation

1: **procedure** CALCUATESUMMARY($f, body, O$: *PathOracle*) ▷ The summary of method f
2: $result\_sum \leftarrow \phi$, $sum \leftarrow \{\phi, ((f, \texttt{Entry}), \phi, \phi)\}$
3: $cnt \leftarrow 0$, $visit \leftarrow \phi$
4: **while** $sum \neq \phi \wedge$ unvisited nodes exist $\wedge$ $iter < threshold$ **do**
5: $sum_{\mathsf{pop}}, sum \leftarrow \mathsf{pop\_one\_local\_path}(sum)$ ▷ Depth-first Traversal
6: $visit \leftarrow$ each $sum_{\mathsf{pop}}{}'$s last node
7: $iter = iter + 1$
8: $sum' \leftarrow \{s' \mid s \in sum_{\mathsf{pop}}, s' \in [\![\mathsf{last}(s.\mathsf{p})]\!]_{\mathsf{new}}(s)\}$ ▷ Evaluate semantics
9: $sum'' \leftarrow$ ORACLEPATHFILTER(sum', O) ▷ Select paths through OPF
10: $sum \leftarrow sum \cup$ WEIGHTPATHFILTER(sum'') ▷ Select top-ranked paths through WPF
11: $terminated, sum \leftarrow \mathsf{pop\_terminated}(sum)$
12: $result\_sum \leftarrow result\_sum \cup terminated$
13: **end while**
14: $result\_sum \leftarrow$ WEIGHTPATHFILTER($result\_sum$) ▷ Select top-ranked result paths
15: **return** *result_sum* ▷ The summary of method f is result_sum
16: **end procedure**

OWASP Benchmark[1] is a benchmark to evaluate precision and recall of a vulnerability scanner. It consists of 2740 test-cases which may have one of 10 top vulnerabilities. The ratio of the number of safe test-cases to the number of unsafe test-cases is almost one to evaluate both precision and recall of a vulnerability scanner.

F-measure is a measure combining precision and recall. A F-measure is the harmonic mean of precision and recall. The F-measure is 100 when a scanner's both precision and recall are 100%. The F-measure is 0 when a scanner's both precision and recall are 0%.

[1] www.owasp.org/index.php/Benchmark, v1.2 beta

Baseline analyzer gets F-measure 96 on OWASP Benchmark. An analyzer by our method may get high precision easily because of no restriction on symbolic state design. Table 1 shows that Baseline analyzer achieves precision 100%. Although there are 98 false-negatives, it achieves high recall 93%. The table shows that the introduced path navigation strategy is effective to scan vulnerabilities.

Modified analyzer gets the same F-measure as Baseline analyzer. The experiment result of Modified analyzer is exactly same as Table 1. Section 5.2 shows that the number of explored path reduced by 3.4 times. Nonetheless, the precision and recall are preserved.

Table 1: Evaluation of the path-sensitive analysis technique on OWASP Benchmark (TP=true positive, FN=false negative, TN=true negative, FP=false positive)

Category	Test query		Result				Score		
	unsafe	safe	TP	FN	TN	FP	recall(%)	precision(%)	F-measure
Command Injection	126	125	126	0	125	0	100	100	100
Weak Cryptography	130	116	97	33	116	0	75	100	85
Weak Hashing	129	107	89	40	107	0	69	100	82
LDAP Injection	27	32	27	0	32	0	100	100	100
Path Traversal	133	135	123	10	135	0	92	100	96
Secure Cookie Flag	36	31	36	0	31	0	100	100	100
SQL Injection	272	232	272	0	232	0	100	100	100
Trust Boundary Violation	83	43	83	0	43	0	100	100	100
Weak Randomness	218	275	203	15	275	0	93	100	96
XPATH Injection	15	20	15	0	20	0	100	100	100
XSS	246	209	246	0	209	0	100	100	100
Total	1415	1325	1317	98	1325	0	93	100	96

5.2. Experiment 2: Improvement of Efficiency

We construct a new benchmark containing up to one million lines of code to evaluate the pre-analysis technique. They are listed in Table 2. Two of them: sparrowml and objchk are our internal java program. Other ten programs are open source projects composed of Java and jsp. Especially four of them: josso1, jdesurvey, broadleaf commerce, and alfresco are open source projects using Spring framework. The largest number of lines of code in the benchmark has about one million lines of code.

Baseline analyzer efficiently covers vulnerable paths. The size of codes in OWASP Benchmark is 161K LOC. it analyzes the codes within 10 minutes. This is 1 Million LOC/hour phase.

Our experimental result presents the pre-analysis technique makes Baseline analyzer several times more efficient. In spite of additional pre-analysis, the total analysis time is reduced by several times. The analysis time of Modified analyzer dramatically decreases comparing to Baseline analyzer. It is reduced by about 6 times. The total analysis time is also reduced by 2.5 times. Many programs in the benchmark are analyzed about 4 times faster. Even javassist having the least effect is analyzed about 1.6 times faster.

Table 2 shows that additional true-positives are discovered by the new method. Although, the increased number of true-positives is not so large, but it is a meaningful number in comparison to the decreased number. Only tomcat loses one true-positive. The others lose nothing. There are no declines for josso1 when 38 new vulnerabilities are detected. Even tomcat who lose one true-

positive gain 9 additional true-positives. Reduction of explored paths makes analysis time decrease, but rather precision increase.

Table 2: Evaluation of the pre-analysis technique on 2 internal private source projects and 10 open source projects.

LOC	Benchmark (java+jsp)	Explored local path(#)			Analysis time(s)					Alarm Diff	
		prev	new	ratio	prev main	new			ratio	Δ TP	ΔFN
						pre	main	total			
20K	sparrowml	6,134	638	x9.6	138	12	7	19	x7.3	0	0
26K	webgoat	962	337	x2.9	95	10	12	22	x4.3	0	0
32K	objchk	6,901	847	x8.1	181	15	25	40	x4.5	0	0
40K	goatdroid	5,470	1,137	x4.8	67	17	13	30	x2.2	0	0
65K	josso1	9,195	2,161	x4.3	430	44	49	93	x4.6	38	0
79K	javassist	9,022	1,991	x4.5	155	84	15	99	x1.6	0	0
83K	roller	12,277	4,904	x2.5	268	37	49	86	x3.1	3	0
88K	jdesurvey	3,075	2,358	x1.3	277	23	40	63	x4.4	0	0
156K	struts	22,085	5,847	x3.8	689	147	110	257	x2.7	0	0
196K	broadleaf c.	25,911	7,140	x3.6	1,618	161	228	389	x4.2	4	0
249K	tomcat	47,156	13,021	x3.6	1,589	197	205	402	x4.0	9	1
969K	alfresco	107,839	35,399	x3.0	7,384	2,291	1,454	3,745	x2.0	1	0
	TOTAL	256,027	75,780	x3.4	12,891	3,038	2,207	5,245	x2.5	55	1

The number of detections of deep inter-path vulnerable paths increases. The paths of the additional true positives have more than 3 call-depth. The deepest inter-path has 5 call-depth with long inter-procedural path. Baseline analyzer detects long inter-paths which include many call-sites frequently, but paths who have deep inter-paths rarely.

Another encouraging point is that the number of vulnerabilities detected among nested branches increases.

5.3. Discussion

Our path-sensitive analysis technique is scalable and has ability to achieve high precision and recall. Table 2 shows that an analyzer implemented by only the path-sensitive analysis technique is able to analyze 1 million LOC. Table 1 shows that the analyzer has high precision and recall.

The pre-analysis technique is a goal-oriented path navigation technique. Goals are clearly reflected by Path-oracle in the algorithm. Path-oracle guided path navigation saves unnecessary path searches. The detection rate of vulnerabilities having deep inter-paths or many branching increases.

Reduction of the number of explored paths leads the time reduction. Although the reduced number of explored paths and the reduced time are not exactly proportional, but most of them are highly correlated. As an unusual case, jdesurvey's number of explored path is reduced by only 1.3 times when the time reduced by 4.4 times. The Modified analyzer avoided paths that are meaningless but time-consuming.

Additional false-negatives may be raised because of two reasons. First, the proposed algorithm calculates approximate fixed point of a Pre-analysis semantics. A Pre-analysis may miss vulnerable nodes. Second, the heuristic for effective interval is not optimal. A Pre-analysis may fail to find infeasible paths.

A disadvantage of the pre-analysis technique occurs when modifying the design of target analysis. Simultaneous modifications of a Pre-analysis semantics and the Main-analysis semantics are not always necessary, but a developer should always examine whether modification of the Pre-analysis semantics is necessary or not. Changes in core design of a Main-analysis require a design change in the Pre-analysis. An incorrect Pre-analysis makes the Main-analysis lose opportunity to detect vulnerable paths.

6. FUTURE WORK

A more researches for heuristics to find effective intervals are required. The effective interval classifier works using heuristics rather than logical methods. We do not know our heuristics are close to the optimal although the heuristics are effective. How to make effective interval classifier is a key issue left.

Effective interval classifier trained by machine learning algorithm may be used. [10] presents a method for learning a classifier which select good analysis parameters of a target code via Bayesian optimization. The advantage of this method is that Bayesian optimization do not require training data but just an objective function which is path-sensitive constant propagation in this case. Supervised learning algorithm like SVM requires training data and obtaining the training data, effective intervals in this case, is not easy.

We have a plan to improve effective interval classifier using Bayesian optimization. Feature vector is a crucial factor for effective learning. Most of the research will be finding the good feature vector. We will look at many codes and study important features for effective interval.

7. CONCLUSIONS

We have given a scalable path-sensitive analysis technique for many types of security vulnerabilities with high precision and recall. The technique do not restrict design condition of symbolic state for making highly precise vulnerability scanner to be easy. Scalability is achieved by a path navigation heuristic only. Experimental results show that it can analyze 1 million lines of code. Despite the path navigation heuristic filters out many paths for scalability, it does not lose recall.

We have presented a pre-analysis technique improves the efficiency of the path-sensitive analysis. Both analysis time and recall are improved by the method. Experimental results show that our technique reduces the number of explored paths by 3.4 times. Due to the reduction, the analysis time reduced by 2.5 times in spite of the additional cost for a pre-analysis. Despite the number of explored paths is reduced, false negatives are reduced.

ACKNOWLEDGEMENTS

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP). (No.R0190-15-1099, Development of an integrated management system and a security testing system that enables interaction between security vulnerability detection technologies in development and operation phases of web application)

REFERENCES

[1] Balakrishnan, G., Sankaranarayanan, S., Ivani, F., Wei, O., and Gupta, A. (2008). Slr: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In Static Analysis. SAS 2008. Lecture Notes in Computer Science, vol 5079. Springer.

[2] Gizem, Aksahya & Ayese, Ozcan (2009) Coomunications & Networks, Network Books, ABC Publishers.

[3] Cousot, P. and Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceeding POPL '77 Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. ACM.

[4] Das, M., Lerner, S., and Seigle, M. (2002). Esp: Pathsensitive program verification in polynomial time. In Proceeding PLDI '02 Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation. ACM.

[5] Dillig, I., Dillig, T., and Aiken, A. (2008). Sound, complete and scalable path-sensitive analysis. In PLDI '08 Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM.

[6] Gutzmann, T., Lundberg, J., and Lowe,W. (2007). Towards path-sensitive points-to analysis. In Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007). IEEE.

[7] Jaffar, J., Murali, V., Navas, J. A., and Santosa, A. E. (2012). Path-sensitive backward slicing. In Proceeding SAS'12 Proceedings of the 19th international conference on Static Analysis. ACM.

[8] Le, W. and Soffa, M. L. (2008). Marple: A demand-driven path-sensitive buffer overflow detector. In Proceeding SIGSOFT '08/FSE-16 Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. ACM.

[9] Li, H., Kim, T., Bat-Erdene, M., and Lee, H. (2013). Software vulnerability detection using backward trace analysis and symbolic execution. In 2013 International Conference on Availability, Reliability and Security. IEEE.

[10] Navabi, A., Kidd, N., and Jagannathan, S. (2010). Pathsensitive analysis using edge strings. In DEPARTMENT OF COMPUTER SCIENCE TECHNICAL REPORTS. Purdue University.

[11] Oh, H., Yang, H., and Yi, K. (2015). Learning a strategy for adapting a program analysis via bayesian optimisation. In OOPSLA 2015 Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM.

[12] Santelices, R. and Harrold, M. J. (2009). Spa: Symbolic program approximation for scalable path-sensitive analysis. In CERCS Technical Reports [193]. Georgia Institute of Technology.

[13] Vojdani, V. and Vene, V. (2009). Goblint: Path-sensitive data race analysis. In Annales Univ. Sci. Budapest., Sect. Comp.

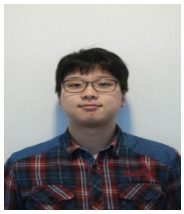
[14] Winter, K., Zhang, C., Hayes, I. J., Keynes, N., Cifuentes, C., and Li, L. (2013). Path-sensitive data flow analysis simplified. In Formal Methods and Software Engineering. ICFEM 2013. Lecture Notes in Computer Science, vol 8144. Springer.

[15] Xie, Y. and Aiken, A. (2005). Context- and path-sensitive memory leak detection. In ESEC/FSE-13 Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering. ACM.

[16] Zheng, Y. and Zhang, X. (2013). Path sensitive static analysis of web applications for remote code execution vulnerability detection. In Proceeding ICSE '13 Proceedings of the 2013 International Conference on Software Engineering. IEEE.

AUTHORS

Dongok Kang



Working at Fasoo.com R&D Center as an engineer

M.S. and B.S in Computer Science and Engineering from Seoul National University

Minsik Jin

Working at Fasoo.com R&D Center as a team leader.

M.S. and B.S. in Computer Science and Engineering from Seoul National University