Computers & Security 128 (2023) 103121

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Insecurity Refactoring: Automated Injection of Vulnerabilities in Source Code

Felix Schuckert[a,b,*], Basel Katt[a], Hanno Langweg[a,b]

[a] NTNU, Norwegian University of Science and Technology, Faculty of Information Technology and Electrical Engineering, Department of Information Security and Communication Technology, Gjøvik Norway
[b] HTWG Konstanz University of Applied Sciences, Department of Computer Science, Konstanz Germany

A R T I C L E   I N F O

Article history:
Received 30 April 2021
Revised 21 December 2022
Accepted 24 January 2023
Available online 27 January 2023

Keywords:
Web security
Static code analysis
Refactoring
Vulnerability Pattern
PHP
SQLi
XSS

A B S T R A C T

Insecurity Refactoring is a change to the internal structure of software to inject a vulnerability without changing the observable behavior in a normal use case scenario. An implementation of Insecurity Refactoring is formally explained to inject vulnerabilities in source code projects by using static code analysis. It creates learning examples with source code patterns from known vulnerabilities.

Insecurity Refactoring is achieved by creating an Adversary Controlled Input Dataflow tree based on a Code Property Graph. The tree is used to find possible injection paths. Transformation of the possible injection paths allows to inject vulnerabilities. Insertion of data flow patterns introduces different code patterns from related Common Vulnerabilities and Exposures (CVE) reports. The approach is evaluated on 307 open source projects. Additionally, insecurity-refactored projects are deployed in virtual machines to be used as learning examples. Different static code analysis tools, dynamic tools and manual inspections are used with modified projects to confirm the presence of vulnerabilities.

The results show that in 8.1% of the open source projects it is possible to inject vulnerabilities. Different inspected code patterns from CVE reports can be inserted using corresponding data flow patterns. Furthermore the results reveal that the injected vulnerabilities are useful for a small sample size of attendees (n=16). Insecurity Refactoring is useful to automatically generate learning examples to improve software security training. It uses real projects as base whereas the injected vulnerabilities stem from real CVE reports. This makes the injected vulnerabilities unique and realistic.

1. Introduction

Automating the injection of vulnerabilities into a codebase can yield valuable knowledge for two cases: How easy is it for an attacker to quickly add vulnerabilities in a short period of time? This is a scenario that could be observed in attacks on PHP Git repositories where a backdoor was inserted PHP repository - backdoor commit (2021). A second case is training of software developers for inspections by the help of complex code samples. Oftentimes, training samples containing vulnerabilities are manually created Du (2011). This requires a significant effort and usually leads to small applications built around few vulnerabilities. Software inspections in the field, however, deals with large and complex applications where vulnerabilities are not easy to spot. Using existing applications with known vulnerabilities is insufficient for training situations, because learners are able to find the vulnerabilities documented in publicly available databases. Hence, automatically generated vulnerabilities have been proposed, e.g., by Stivalet and Fong (2016) and Boland and Black (2012). Those automatically generated samples are artificial and can be used to benchmark tools for static code analysis.

Our approach is to automatically create learning examples by modifying existing large projects. To achieve that, we use vulnerability patterns to inject vulnerabilities into open source projects. The use of existing projects ensures that the context of a vulnerability is as real as possible. We created source code patterns by examining vulnerabilities and corresponding fixes in source code spanning a period of multiple years Schuckert et al. (2017, 2019). The source code originated from real applications for which vulnerabilities had been reported with an assigned CVE-ID (CVE: Com-

* Corresponding author.
E-mail address: felix.schuckert@htwg-konstanz.de (F. Schuckert).

mon Vulnerabilities and Exposures). Our contribution answers the following questions:

- Are static code analysis and refactoring valid approaches to inject vulnerabilities in existing projects?
- How can source code patterns from recent vulnerabilities be represented within the injected vulnerabilities?
- Are the insecurity refactored applications useful for teaching software security?

Insecurity Refactoring is a change to the internal structure of software to inject a vulnerability without changing the observable behavior in a normal use case scenario. We proposed a technique to conduct insecurity refactoring using static code analysis methods. An Adversary Controlled Input Dataflow (ACID) tree is constructed to find possible injection paths. These possible injection paths are then transformed into vulnerabilities using patterns from known vulnerabilities. The implementation is evaluated on open source projects to find possible injection paths and inject vulnerabilities.

Section 2 provides an overview of work related to Insecurity Refactoring and describes the definition of the Code Property Graph. Section 3 defines and formulates the methods and concepts proposed in our methodology. Section 4 formulates the tree construction mechanism that is used for the Insecurity Refactoring process. The definitions of Insecurity Refactoring are described in Section 5. Section 6 describes the source code pattern language PL/V and how vulnerabilities are injected. Followed by Section 7, in which the approach is evaluated on open source projects from GitHub. In this context the usefulness of the methodology in software security training is evaluated by an experiment involved two groups with different skill levels. The final Section 8 points out problems and concerns of Insecurity Refactoring.

2. Background

Thomas et al. (2009) developed a tool that replaces database queries with prepared statements to remove potential SQL Injection vulnerabilities. They present a prepared statement replacement algorithm (PSR-Algorithm) that separates the SQL query string from any input strings. They evaluated their tool on IT security training projects like WebGoat and 94% of their refactored prepared statements prevented SQL Injection attacks. Maruyama and Omori (2011) present a security-aware refactoring tool. Normal refactoring approaches can create unintended security issues. The focus relies on accessibility of class variables. Refactoring approaches can change the accessibility without changing the external behavior that might result in a security issue. Their tool actually checks for such security issues and provides refactoring approaches that do not create such issues.

Dolan-Gavitt et al. (2016) created a tool named Large-scale Automated Vulnerability Addition (LAVA) that uses dynamic taint analysis to find locations to inject vulnerabilities in C/C++ projects. The dynamic approach makes the data flow analysis easier. The injected vulnerabilities themselves are artificial. Nevertheless, injecting vulnerabilities in real projects provides a more realistic scenario than manually creating a small project that contains a vulnerability. Also a LAVA-M data set was released that contains many injected vulnerabilities in C-projects. That data set is commonly used to evaluate modern fuzzers Klees et al. (2018) Rawat et al. (2017).

Pewny and Holz (2016) developed the EvilCoder tool similar to LAVA that injects bugs in C-projects. Similar to this work, the Code Property Graph is used to find potential injection locations. The focus relies on memory critical functions and corresponding security checks. These security checks are replaced by insufficient checks to create vulnerabilities. An evaluation on four open source projects shows the potential to conduct injections at many different locations. However, they cannot ensure that an injected vulnerability is exploitable.

Our approach uses the Code Property Graph defined by Yamaguchi et al. (2014). The Code Property Graph combines an Abstract Syntax Tree, Control Flow Graph and Program Dependence Graph into a single graph. They use the graph to find vulnerabilities in C/C++ projects. Backes et al. (2017) extended the Code Property Graph to support PHP. We use this PHP graph to create the Adversary Controlled Input Dataflow tree. Alhuzali et al. (2018) also used the Code Property Graph to find vulnerabilities in PHP projects. They added support to automatically create exploits for the discovered vulnerabilities. The results show that the Code Property Graph is very useful to discover vulnerabilities in C/C++ and PHP.

The usage of insecurity refactoring to create learning example seems promising. Schreuders et al. (2017) developed the Security Scenario Generator (SecGen) that allows to create multiple virtual machines containing different vulnerabilities. The vulnerabilities are defined by modules. Based on the module description, vulnerabilities can be nested and hints can be placed. A survey that has been used to evaluate the usage of such generated virtual machines is helpful as learning examples. Yamin and Katt (2022a) Yamin and Katt (2022b) developed a similar framework to automatically create full cyber security ranges that setup multiple virtual machines. The focus relies on creating a large number of virtual machines and the cyber security ranges are used for different scenarios like attacker/defence CTF events. Based on the scenario different vulnerabilities are injected to the machines via ssh. For example, injected vulnerabilities can be weak passwords, misconfigurations, components with known vulnerabilities, etc. Chapman et al. (2014) designed the PicoCTF tool and hosted a capture the flag (CTF) event where approximately 2,000 teams participated. The approach was game based. The tasks can either be viewed in computer game style including a story or in a classical text view. A survey has been used to evaluate the approach. The results show that the approach is useful and many other CTF events have used the PicoCTF tool. Burket et al. (2015) explain the automatic problem generation (APG) for PicoCTF. The APG allows to generate CTF tasks that differ for each attending team. A templated autogen problem uses a fixed template and multiple inputs (e.g. flag) to generate the CTF task. This allows to detect key sharing between teams but does not prevent sharing the *method* to solve the task between teams. In contrast, challenges that are automatically generated without a fixed template have problems with consistent difficulties, bug prevention, scalability and deployment. Another PicoCTF event has been held using a templated autogen to reveal that key-sharing actually exists and can be detected by the approach.

2.1. Code Property Graph

This section explains the definitions introduced by Yamaguchi et al. (2014) of the Code Property Graph (CPG) and traversal functions.

Definition 1. A Code Property Graph $G = (V, E, \lambda, \mu)$ is a directed, edge-labeled and attributed multigraph. V is the set of nodes, $E \subseteq (V \times V)$ is the set of directed edges. The labels of these edges are defined by $\lambda : E \to \Sigma$ where alphabet Σ represents all edge names. Properties for edges and nodes are assigned by $\mu : (V \cup E) \times K \to S$. K is a set of property keys and S is the set of property values.

The Code Property Graph is based on the Abstract Syntax Tree (AST). The Abstract Syntax Tree is defined as follows:

$$G_A = (V_A, E_A, \lambda_A, \mu_A) \tag{1}$$

The Abstract Syntax Tree has one kind of edge labels (*parent_of*), which is defined in the set λ_A. The set μ_A contains property assignments for every node of the Abstract Syntax Tree. For example, the name of a variable is stored as a property and the value is the variable name.

The Control Flow Graph is defined as follows:

$$G_C = (V_C, E_C, \lambda_C, \varnothing) \tag{2}$$

The nodes $V_C \subseteq V_A$ are statements of the programming language. For example, an assignment is a statement. Edges E_C represent the possible control flow from a statement to another statement. For the edges, only one kind of label (*flows_to*) exists that is defined in λ_C. No properties are stored for the Control Flow Graph.

The Program Dependence Graph defines where variables are used and resolves function calls. The definition for the Program Dependence Graph is as follows:

$$G_P = (V_P, E_P, \lambda_P, \mu_P) \tag{3}$$

The nodes $V_P \subseteq V_A$ are the same nodes as of the Abstract Syntax Tree. Edges E_P either represent function calls or variable usage. Function calls have the edge label *calls*. The variable usages have the label *reaches* that point from the variable definitions to the statements where the variables are used. These edge labels are defined in λ_P. For the *reaches* edges the property μ_P defines the variable name of the variable definition.

Definition 2. A traversal is defined as a function $\tau : \mathbb{P}(V) \to \mathbb{P}(V)$ that maps a set of nodes to another set of nodes according to a Code Property Graph G, where $\mathbb{P}(V)$ is the power set of V.

The following function definition allows to iterate over an edge:

$$OUT_l(X) = \bigcup_{v \in X} \{u : (v, u) \in E \land \lambda((v, u)) = l\} \tag{4}$$

$$OUT_l^{k,s}(X) = \bigcup_{v \in X} \{u : (v, u) \in E \land \lambda((v, u)) = l \land \mu((v, u), k) = s\} \tag{5}$$

$$IN_l(X) = \bigcup_{u \in X} \{v : (v, u) \in E \land \lambda((v, u)) = l\} \tag{6}$$

$$IN_l^{k,s}(X) = \bigcup_{u \in X} \{v : (v, u) \in E \land \lambda((v, u)) = l \land \mu((v, u), k) = s\} \tag{7}$$

where $X \subseteq V$ is a set of nodes. *OUT* and *IN* return all reachable nodes with the label l and property with key k and value s. The *OUT* function follows the direction of the edge and the *IN* function is a backward iteration of the edge.

We use following functions:

$$Filter_p(X) = \{v \in X : p(v)\} \tag{8}$$

$$Match_p(X) = Filter_p \circ TNodes(X) \tag{9}$$

$$Types_s(X) = TypeNode \circ Filter_{p_s} \circ TNodes \tag{10}$$

$$Stmt(X) = Statement(X) \tag{11}$$

The *Filter* function returns all nodes of the set X that match the Boolean predicate $p(v)$. *TNodes* is defined as a reusable traversal from the root of the Abstract Syntax Tree to all nodes. The $Match_p$ uses the *TNodes* function to traverse all Abstract Syntax Tree nodes and only returns the nodes that match the filter function. The $Types_s$ function iterates the children of the Abstract Syntax Tree starting from node $x \in X$ searching for nodes of the type s. The $Statement(X)$ functions iterates the parents nodes until it reaches a statement node. This is important to get the statement where a specific node $x \in X$ is used. We use the short name *Stmt* instead of *Statement*.

3. Methodology

The goal of Insecurity Refactoring is to inject vulnerabilities with different source code patterns into existing projects. This approach is based on static code analysis concepts. Figure 1 shows the process to inject vulnerabilities. The Code Property Graph Yamaguchi et al. (2014) is used as an initial analysis model. Rules defined in the next section are applied to traverse the Code Property Graph to create the Adversary Controlled Input Dataflow (ACID) tree. The ACID tree is a tree representation of a backward data flow analysis. In the tree, a path from a leaf to the root represents data flow from a source to a sink. It is used as another analysis model to find Possible Injection Paths (PIP) or vulnerabilities. A vulnerability is basically a path (leaf to root) in the tree that does not contain any sanitization functions. In contrast, a PIP does contain a sanitization function. A PIP can be transformed to fit a vulnerability definition. For example, a sanitization method can be refactored into an insufficient sanitization method. To define insufficient sanitization methods, the context of the input data is analyzed using the context rules. The modifications are based on source code patterns. These patterns are defined in the PL/V pattern language that is described in Section 6.1. Additional source code patterns can be injected by using the data flow patterns to add some diversification. The ACID tree uses a tree structure based on nodes from the Abstract Syntax Tree. The Abstract Syntax Tree is an abstract representation of the source code. Refactoring is applied to the Abstract Syntax Tree to create a modified Abstract Syntax Tree. In the last step, the modified Abstract Syntax Tree is used to generate the insecurity-refactored source code.

3.1. Adversary Controlled Input Dataflow Tree

The first step for Insecurity Refactoring is to find PIPs. A PIP is a set of source code statements that can be refactored to inject a vulnerability. This approach focuses on vulnerabilities that have tainted data controlled by an adversary flowing from a source to a sink. Examples of vulnerabilities for this type are Cross Site Scripting (CWE-79), SQL Injection (CWE-89), Buffer Overflow (CWE-119), etc. Our approach uses a modified backward taint analysis. A normal taint analysis stops and removes any data that reaches sanitization methods. The modified taint analysis does not remove such data, instead it further tracks the data. This allows, in the refactoring step, to remove or modify the sanitization method to inject a vulnerability.

Data flow analysis can either be done forward (from source to sink) or backward (from sink to source). We use a backward data flow analysis using a Code Property Graph and follow specified rules to create an Adversary Controlled Input Dataflow tree. The idea is to use a backward data flow analysis following each path of data that flows into the initial sink. These paths are represented in a tree, where the root is the sink of a vulnerability. Each leaf represents data that can reach the sink. Accordingly, a leaf represents a source. An advantage of creating a tree using backward data flow analysis is that the ACID tree allows analyzing all data concatenations that reach a sink. Every leaf represents possible input to reach the sink, but it doesn't necessarily mean that all of the leaves are concatenations.

Definition 3. An *Adversary Controlled Input Dataflow* (ACID) tree

$$T_{AC} = (V_{AC}, E_{AC}, \lambda_{AC}, \mu_{AC}) \tag{12}$$

is an ordered, rooted, directed, edge-labeled and attributed out-tree Deo (1974). The nodes of the tree are defined in the set $V_{AC} \subseteq V_A$. Accordingly, the tree is based on Abstract Syntax Tree nodes and each node can be used to access the corresponding abstract syntax sub tree G_A from the Code Property Graph. The di-

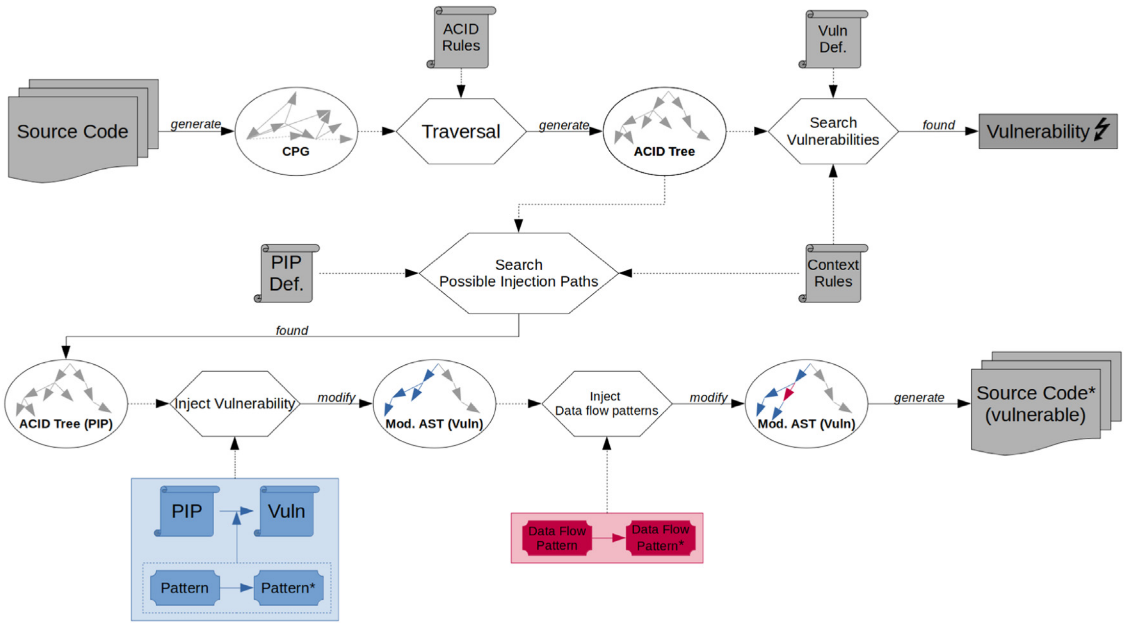


Fig. 1. Overview of the Insecurity Refactoring process by using an ACID tree.

rected edges are defined as the set $E_{AC} \subseteq (V_{AC} \times V_{AC})$. The edge label function $\lambda_{AC} : E \rightarrow \Sigma_{AC}$ uses the alphabet Σ_{AC} to represent all edge names. The properties for the nodes are defined by the function $\mu_{AC} : V_{AC} \times K_{AC} \rightarrow S_{AC}$. The set K_{AC} defines the keys and the set S_{AC} defines the values. All attributes from the initial Abstract Syntax Tree nodes are found in the ACID tree nodes. Because the tree is ordered, similar to the Code Property Graph, we add an attribute with the key *childN* that stores the child position as a value.

The **root node** of an ACID tree is defined by:

$$\begin{aligned} root_{AC} &= root(V_{AC}, E_{AC}) \\ &= u \in V_{AC} : \nexists(\bullet, u) \in E_{AC} \end{aligned} \tag{13}$$

It returns the node, to which there are no directed edges pointing.

Additionally, to get all **leaf nodes** of an ACID tree, we define the following function:

$$\begin{aligned} L_{AC} &= L(V_{AC}, E_{AC}) \\ &= \{v \in V_{AC} : \nexists(v, \bullet) \in E_{AC}\} \end{aligned} \tag{14}$$

It returns all nodes that do not have an edge pointing to other nodes.

The **children** of a node v can be retrieved with the following definition:

$$C(v) = \{u : (v, u) \in E_{AC}\} \tag{15}$$

It returns a completely ordered set. The order is defined by the children positions in the tree.

Also, the **parent** of a node u can be retrieved with the following function:

$$p(u) = v \in V_{AC} \text{ where } (v, u) \in E_{AC} \tag{16}$$

The function

$$Path(l) = \langle l \rangle ^\frown Path(p(l)) \tag{17}$$

defines a sequence of nodes starting from the leaf node l going upwards until reaching the root node $root_{AC}$ of the ACID tree.

The siblings of a node can be retrieved with the following functions:

$$Sib(v) = \{u : p(v) = p(u) | u \neq v\} \tag{18}$$

$$Bef(v) = \{u : p(v) = p(u) | \mu(u, childN) < \mu(v, childN)\} \tag{19}$$

$$Aft(v) = \{u : p(v) = p(u) | \mu(u, childN) > \mu(v, childN)\} \tag{20}$$

This allows to get all siblings (Sib), the siblings before (Bef), or the siblings after (Aft) a node v.

The edge labels are used to specify the data type that flows from one node to another. The data types are defined in the alphabet $\Sigma_{AC} = \{String, Numeric, Array, Unknown\}$. The properties for the nodes are defined in τ. We use the properties to define the different splits in the ACID tree. A split in the ACID tree means that either data from all sub trees will reach the sink, or only one sub tree at a time can reach the sink. The *link* property defines the link between children. The values are from the alphabet $\tau = \{\wedge, \oplus\}$. *Excluding* is defined by the symbol $\oplus$. It means that either one of the sub trees will reach the sink. A *concatenation* is defined by the symbol $\wedge$. It indicates that a concatenation of the children will reach the sink. We use the *cap* symbol, since every child has to add its input to the concatenation. For *excluding*, we assume that both paths are reachable in certain instances. Control statements decide which sub trees will reach the sink, i.e., they check code reachability.

3.2. Code example

Figure 2 shows a code example that is used to describe the process of Insecurity Refactoring. On line 7, the *getParam()* function is used to request the page number from the user. The page number is checked for being numeric (line 14) and sanitized using the *intval* (line 15) function. Hence, no Cross Site Scripting attacks are possible.

4. ACID Tree Construction

The ACID tree is constructed by traversing the Code Property Graph. An ACID tree is created for each potential sink. The traversal requires a stack $stack_{call}$ that is used to correctly resolve function calls. The stack $stack_{call}$ stores the function calls that are resolved by the traversal. The traversal is based on different node types. The main traversal in the Code Property Graph is over the

```php
<?php
function getParam($param){
  return $_GET[$param];
}

function page($debug, $name){
   $page=getParam('page');



   if(is_numeric($page)){
      $out = $name . intval($page);
      $out = "<a href='www.url.com/" . $out .
           "'> link </a>";
   }
   else {
      $out = "Unknown page";
   }

   echo $out;
}
?>
```

Fig. 2. Code example shows proper sanitization (line 14 and 15) to prevent Cross Site Scripting.

V_P nodes from the program dependence graph. We define the following node categories that are used by the Abstract Syntax Tree G_A and the program dependence graph G_P:

$$V_{assign} = \{v \in V_P \; | v \in V_A \\ \quad |\mu_A((v,u), type) = assignment\} \tag{21}$$

$$V_{param} = \{v \in V_P | v \in V_A \\ \quad |\mu_A((v,u), type) = parameter\} \tag{22}$$

Edges (E_P) point from a variable definition to statements (V_P) where the defined variable is used. Because the traversal is backwards, the definitions are traversed. A definition can either be an assignment ($v \in V_{assign}$) or it can be a function parameter ($v \in V_{param}$).

For the Control Flow Graph, the following node categories are important:

$$V_{function} = \{v \in V_C \; | v \in V_A \\ \quad |\mu_A((v,u), type) = function\} \tag{23}$$

The traversal also traverses concatenations, variables, function calls and coding constructs. We define the following sets to represent these node categories:

$$V_{exp} = \{v \in V_A \text{ that represent all expression}\} \tag{24}$$

$$V_{var} = \{v \in V_{expr} | \mu_A(v, type) = variable\} \tag{25}$$

$$V_{con} = \{v \in V_{expr} | \mu_A(v, type) = concatenation\} \tag{26}$$

$$V_{call} = \{v \in V_{expr} | \mu_A(v, type) = call\} \tag{27}$$

$$V_{code} = \{v \in V_{expr} | v \text{ is a coding construct}\} \tag{28}$$

The set V_{exp} contains all expressions. An expression always has a return value. The traversal distinguishes expressions between variables V_{var}, concatenations V_{con}, function calls V_{call} and coding constructs V_{code}. For simplicity, all different concatenations that exist are unified by having the type *concatenation*. Coding constructs are different constructs that depend on the programming language. For example, in our implementation, the array attribute access is included. In the Abstract Syntax Tree, it is represented by a *dimension* node. The traversal is based on the different categories. For each powerset $\mathbb{P}$ of a category, the corresponding function is used to traverse the Code Property Graph. The following function

$$Pos : V_A \to \mathbb{N} \\ Pos(v) = p \in \mathbb{N} | \mu(v, childnum) = p \tag{29}$$

allows getting a position of parameter. The position n can be used to get an expression of a function call with the following function:

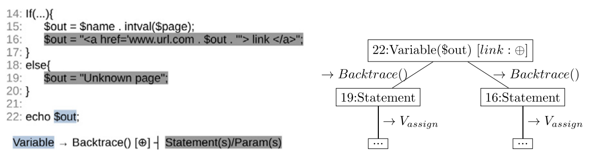
$$CallExp(V, n) = OUT^{childnum,n}_{parent\_of}(V) \tag{30}$$

These functions are used in the ACID tree construction to correctly resolve function calls.

The following function defines **backward traversal** for the Code Property Graph:

$$Backtrace : \mathbb{P}(V_{var}) \to \mathbb{P}(V_{assign} \cup V_{param}) \\ Backtrace(V) = \bigcup_{v \in V} \{IN^{variable,v}_{reaches}(Stmt(\{v\}))\} \tag{31}$$

It uses the variable as input and returns the corresponding nodes where the variables are defined. It uses the *Stmt* function to get the statement where the variable v is used. The statement is used to get possible definitions of the variable v. The results can be assignments or parameters. Figure 3 shows how the *Backtrace*() function is used from the variable *$out*.



```
14: If(...){
15:      $out = $name . intval($page);
16:      $out = "<a href='www.url.com . $out . "'> link </a>";
17: }
18: else{
19:      $out = "Unknown page";
20: }
21:
22: echo $out;

Variable → Backtrace() [⊕] ┤ Statement(s)/Param(s)
```

(a) Source code. (b) ACID tree.

Fig. 3. Rules example of *Backtrace*() using the program dependence graph.

In the ACID tree, these statements are added as children and the ⊕ defines that the children are mutually excluding. Accordingly, only one of the sub trees can reach the sink. For the variable, possible definitions are on line 19 and line 16.

Based on the resulting statement type, the graph is traversed differently. If the statement is an assignment (V_{assign}), the following traversal rules apply to **resolve assignments**:

$$Assign : \mathbb{P}(V_{assign}) \to \mathbb{P}(V_{exp}) \\ Assign(V) = OUT^{childnum,1}_{parent\_of}(V) \tag{32}$$

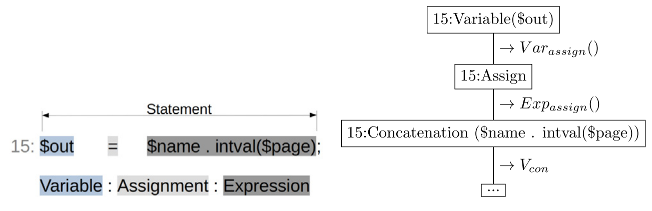
The Abstract Syntax Tree G_A is an ordered tree and the attribute *childnum* is used to define the order. The child number one is the expression that will be assigned. Because the ACID tree is constructed by a backward data flow analysis, the defined variable is added to the ACID tree first, then the assignment statement is added that is followed by the expression from *Assign*(). The defined variable is given by the *OUT* function using the child number zero.

Figure 4 shows how the rules are applied to line 15.

Variable *$out* is added first, it is followed by the assignment node ($v \in V_{assign}$) which is followed by the expression.

The following rules apply to **resolve concatenations**:

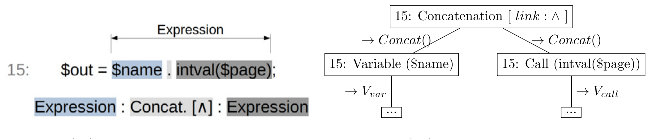
$$Concat : \mathbb{P}(V_{con}) \to \mathbb{P}(V_{exp}) \\ Concat(V) = OUT_{parent\_of}(V) \tag{33}$$



```
15: $out = $name . intval($page);
```

(a) Source code. (b) ACID tree.

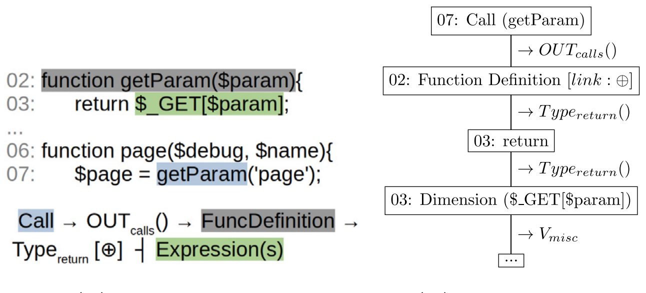
Fig. 4. Rules example of an assignment.



```
15:   $out = $name . intval($page);
```

(a) Source code. (b) ACID tree.

Fig. 5. Rules example of a concatenation.

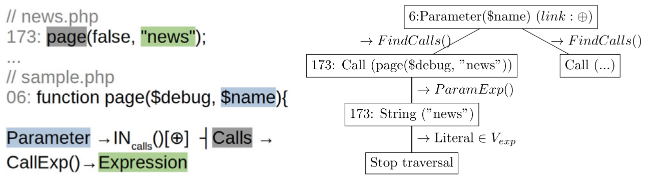


```
02: function getParam($param){
03:     return $_GET[$param];
...
06: function page($debug, $name){
07:     $page = getParam('page');
```

Call → OUT_calls() → FuncDefinition →
Type_return [⊕] ┤ Expression(s)
07: Call (getParam)
→ $OUT_{calls}()$
02: Function Definition [link : ⊕]
→ $Type_{return}()$
03: return
→ $Type_{return}()$
03: Dimension ($_GET[$param])
→ V_{misc}
...

(a) Source code. (b) ACID tree.

Fig. 6. Rules example of a function call.



```
// news.php
173: page(false, "news");
...
// sample.php
06: function page($debug, $name){
```

(a) Source code. (b) ACID tree.

Fig. 7. Rules example of reaching a parameter with an empty call stack.

The function *Concat* allows to get the concatenated elements. Concatenations in the Abstract Syntax Tree use the children as operands. Accordingly, these are the elements that are concatenated and are provided by the *OUT* function.

Figure 5 shows how the *Concat* function is applied to the top expression that is assigned to the variable *$out*.

The dot symbol is the standard method for string concatenation in PHP. A concatenation means that all inputs reach the sink in a concatenated form. Accordingly, the ∧ symbol is added to the concatenation.

A function call has to be resolved to see if the function passes data from the input (parameter) to the output (return). The Code Property Graph already resolves function calls in G_C by the edges with the label *calls*. The following rules apply to **resolve function calls**:

$$CallReturn : \mathbb{P}(V_{call}) \rightarrow \mathbb{P}(V_{exp}) \tag{34}$$

$$CallReturn(V) = \bigcup_{v \in V} \{Type_{return}(OUT_{calls}(\{v\}))\}$$

$\hookrightarrow$ Side-effect: add call to stack $stack_{call}$

The OUT_{calls} returns the function definitions found in the control property graph. Because the traversal is a backward data flow analysis, further analysis has to continue on the output of the function (return statements). The $Type_{return}$ function finds all return statements inside a function definition. A combination of both functions *Call* and $Type_{return}$ is used in *CallReturn* to resolve function calls for the ACID tree. The resolving function call is put on the stack $stack_{call}$ to correctly resolve parameter expressions.

Figure 6 shows how the *CallReturn* function is applied to the source code example.

The function call *getParam* is passed to the *CallReturn* function. It uses the edges E_C from the Control Flow Graph to find the corresponding function definition of the call on line 2. The $Type_{return}$ function is used to find all possible return statements (line 3) of the function definition. If more than one return statement is found, multiple returns are added as mutually excluding (⊕).

For each of these coding constructs, different approaches are required to correctly continue the traversal. The following definitions **resolve coding constructs**:

$$Code : \mathbb{P}(V_{code}) \rightarrow \mathbb{P}(V_{exp}) \tag{35}$$

$$Code(V) = \bigcup_{v \in V} \bigcup_{p \in P_{code}} \{ASTNode_{in}(p, v), \text{ if } Match_p(v)\}$$

Because different code constructs require different approaches to get the correct input, we use the PL/V pattern language described in Section 6.1. The set P_{code} contains all implemented code construct patterns. The *Match* function is used to check if the Abstract Syntax Tree node v equals the pattern p. The *ASTNode* function returns the corresponding input nodes of the Abstract Syntax Tree.

As previously stated, the *Backtrace* function also returns parameters. Depending on whether a function call is currently resolved or not ($stack_{call} = \emptyset$), the backward data flow analysis has to traverse differently and is defined as follows:

$$Param : \mathbb{P}(V_{param}) \rightarrow \mathbb{P}(V_{exp}) \tag{36}$$

$$Param(V) = \bigcup_{v \in V} \begin{cases} FindCalls(\{v\}), & \text{if } stack_{call} = \emptyset \\ BackToCall(Pos(v)), & \text{otherwise} \end{cases}$$

It is a simple function that decides if the *FindCalls* or *BackToCall* is used to traverse parameters V_{param}. If the stack $stack_{call}$ is not empty, the function $BackToCall : \mathbb{N} \rightarrow V_{exp}$ jumps back to the initial function call found on top of the stack $stack_{call}$. The traversal is continued from the corresponding expression based on the parameter position.

If $stack_{call}$ is empty, the following rules are applied to **find all calls**:

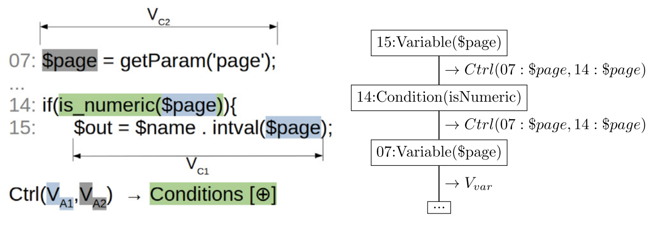
$$FindCalls : \mathbb{P}(V_{param}) \rightarrow \mathbb{P}(V_{exp}) \tag{37}$$

$$FindCalls(V) = \bigcup_{v \in V} \{CallExp(IN_{calls}(\{v\}), Pos(v))\}$$

The *FindCalls* function returns all function calls of the function from the parameter. It uses the function $Pos(p)$ to return the position of the parameter in the function definition. The CallExp function is required to continue the backward data flow analysis from the correct parameter of the function calls.

Figure 7 shows how a parameter is resolved when the stack is empty.

The sample.php is from the code example and one call of function *page* is found in the news.php file on line 173. Additional function calls are added as mutually excluding (⊕) children. The *ParamExp* function uses the parameter position to get back the correct expression of the function call. In the example, the parameter


V_C2
07: $page = getParam('page');
...
14: if(is_numeric($page)){
15: $out = $name . intval($page);
V_C1
Ctrl(V_A1,V_A2) → Conditions [⊕]
15:Variable($page)
→ Ctrl(07 : $page, 14 : $page)
14:Condition(isNumeric)
→ Ctrl(07 : $page, 14 : $page)
07:Variable($page)
→ V_var
...

(a) Source code. (b) ACID tree.

Fig. 8. Rules example of finding condition checks.

is in the second position. Accordingly, the second expression in the function call is returned that is the string literal *"news"*. Because a literal is not found in any of the categories except of V_{exp}, the traversal stops here.

Some function calls can be resolved by the control property graph. Other functions that pass data from a parameter to the return value are not resolved by the control property graph. For example, the *intval* function will not be resolved by the control property graph. To solve that problem, we define the pair $V_{pass} = (F, \delta)$. The set F contains all functions that return data from parameters. The mapping from input to output is defined by the set $\delta \subseteq (F \times \mathbb{N}_0)$. The output of a passthrough function is always the return value. Inputs are parameters that are referenced by the parameter position by a natural number $\mathbb{N}_0$. If the *CallReturn* is unable to resolve the call, the following function will **resolve passthrough** functions:

$$Passthrough : \mathbb{P}(V_{call}) \rightarrow \mathbb{P}(V_{exp}) \tag{38}$$

$$Passthrough(V) = \bigcup_{v \in V} \{CallExp(\{v\}, n) | (v, n) \in \delta\}$$

The function returns the corresponding expression based on the parameter position n.

4.1. Control functions

If we look into the code sample in Figure 2, the security relevant function *is_numeric* will not be traversed. This is a sanitization function call that changes the control flow without changing any data in the data flow. Accordingly, another step is required to find security relevant function calls that change the control flow. Such functions can occur in the traversal of the $Backtrace : \mathbb{P}(V_{var}) \rightarrow \mathbb{P}(V_{assign} \cup V_{param})$ function. For each result, the following functions allow to **find control statements**

$$Ctrl(var_{def}, var_{use}) = \tag{39}$$

$$\bigcup_{path \in Paths(var_{def}, var_{use})} \{ \bigcup_{v \in path} \{Filter_{if}(\{v\}) \circ Match_{var_{def}}(\{v\})\}\}$$

where the function *Paths* returns all possible paths in the Control Flow Graph G_C from variable definition v_{def} to variable usage v_{use}. The $Filter_{if}(c)$ filters only statements that are actually *if* statements. Additionally, the $Match_{var_{use}}(c)$ checks if the initial variable is used inside the *if* statement. Overall, the *Ctrl* function returns all *if* statements where the variable var_{def} is used. In the ACID tree, the control functions are added in between the variable usage and variable definition. The *if* statements themselves are also parsed to correctly handle all conditions that are used. A union in the if statement is a mutually excluding split ($\oplus$) in the ACID tree because only one of the conditions has to be true. For a complement, both conditions are added in serial to the ACID tree because both of the conditions have to be true. Figure 8 shows how the *is_numeric* is found by the condition check.

The *Backtrace* function returns from the variable $*page* on line 15 the assignment on line 7. The *is_numeric* uses the variable $*page* and is found in an *if* statement. Accordingly, it will be added to the ACID tree.

4.2. Data flow type

The construction of the ACID tree requires labeling the edges (λ_{AC}) corresponding to the flowing data type. The flowing data type can only be determined by a forward analysis. Accordingly, the labels are set after the initial ACID tree is constructed. The labels are defined by iterating the nodes of the paths ($Path(l)$) from all leaves of the ACID tree. If a node defines a data type change, the data type changes. For example, the *intval()* function changes the data type to *numeric*. In contrast, a string concatenation changes the data type to string. All nodes that do not change a data type will preserve the previous data type.

4.3. ACID tree example

Figure 9 shows the full ACID tree for the initial code example shown in Figure 2. All leaves are data that can reach the sink. For each node the corresponding line number is added. An edge label represents the data type that flows between the nodes. A simple guide to read an ACID tree is to choose a leaf and go upwards until you reach the root. If you reach a concatenation, the symbol $\wedge$ is shown. It means that the data from the other sub trees of that $\wedge$ node are also included by a concatenation to the sink. In contrast, the $\oplus$ means that only either one of the sub trees reaches the sink. In that case, the other sub trees can be ignored.

In the example, either the *"Unknown page"* string or the *_GET* variable are concatenated ($\wedge$) with the $*name* parameter and the two string values will reach the echo function. The parameter can be different based on what function call is used. In the example, the parameter has the string value *"news"*. The condition functions *isNumeric* and *intval* prevent any Cross Site Scripting attacks.

5. Insecurity Refactoring

Refactoring is defined as a change made to the internal structure of software to make it easier to understand and less expensive to modify without changing its observable behavior Fowler (1999) Mens and Tourwé (2004) Opdyke (1992). Insecurity refactoring uses a similar approach and we define it as: *Insecurity refactoring is a change to the internal structure of software to inject a vulnerability without changing the observable behavior in a normal use case scenario. If the injected vulnerability is exploited, the observable behavior will change.* Accordingly, insecurity refactoring requires to maintain the normal use of the program. The following rules define insecurity refactoring by transforming a PIP into a vulnerability.

5.1. Vulnerability Description

Vulnerabilities can be described in different ways. For example, Martin et al. (2005) used the Program Query Language to describe vulnerabilities. Our focus is on vulnerabilities that rely on data flow from a source to a sink. We define a vulnerability based on three sets V_{src}, V_{dst}, and V_{san}. The sources V_{src} are patterns for retrieval of tainted data. In the code example the $_ GET global array in line 3 is included in the source pattern set (V_{src}) because it provides user-controlled data. The *echo* function in line 22 is a sink contained in the sinks set V_{dst}.

The ACID tree is based on a backward data flow analysis with a sink and an amount of inputs (sources). The inputs are represented by the leaves of the ACID tree.

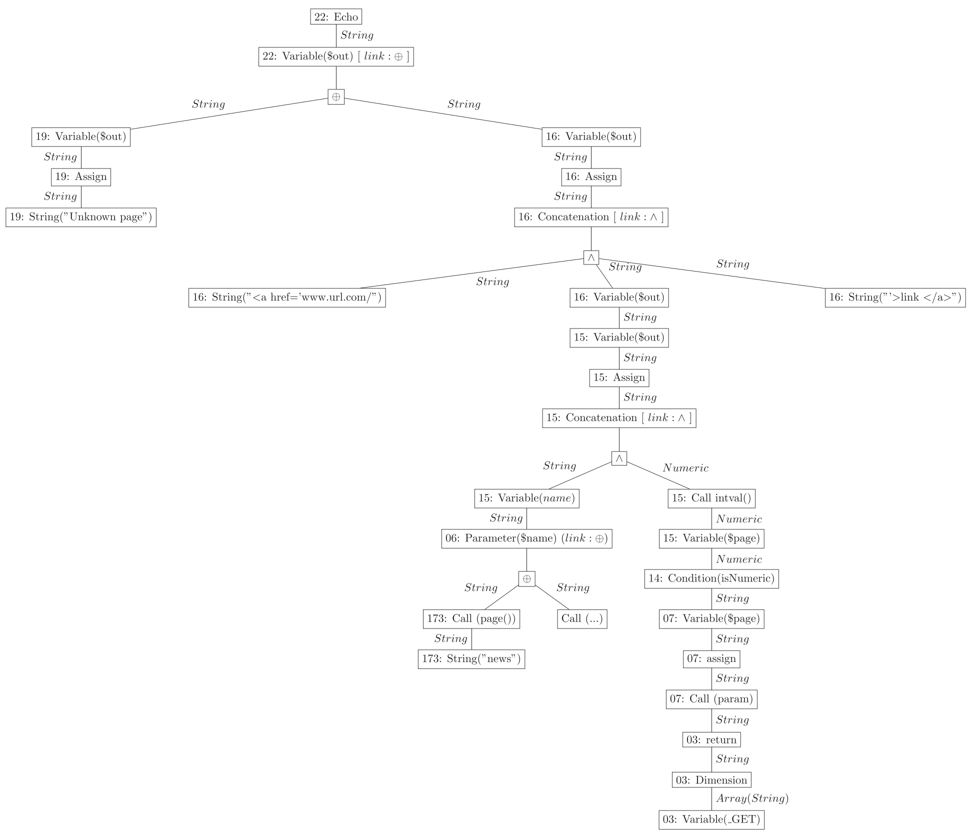


Fig. 9. ACID tree of the code example from Figure 2.

Definition 4. A data flow path $dfp^l_{AC} = (T_{AC}, l)$ is defined as a pair containing the ACID tree and a chosen leaf.

The data flow path represents data that flows from the chosen leaf l into the sink r_{AC}.

The sanitization functions are defined by the set V_{san}. Sanitization functions from V_{san} depend on the context and the vulnerability type. The following function $Suff$ allows to check if a sanitization function v_{san} is sufficient with respect to the data flow path dfp^l_{AC}:

$$Suff(v_{san}, dfp^l_{AC}) = \begin{cases} True, & \exists c : ((v_{san}, root_{AC}), c) \in S \\ & \text{and } c \in Context(dfp^l_{AC}) \\ False, & \text{otherwise} \end{cases} \tag{40}$$

where

$$S \subseteq ((V_{san} \times V_{dst}) \times C_{ctx})$$

The set C_{ctx} contains all possible context types. The set S defines for each sanitization function v_{san} and sink v_{dst} what context c is required. The function $Context()$ returns a set of all active contexts for the data flow path. The details of the function are described in Section 6.2. It returns all contexts that are found in the data flow path dfp^l_{AC}. Accordingly, the function $Suff()$ checks if the sanitization function is sufficient based on the vulnerability type.

Definition 5. The set $Vuln_{AC}$ contains vulnerabilities. A vulnerability $vuln^l_{AC}$ is a data flow path dfp^l_{AC} with the following properties:

$$root_{AC} \in V_{dst} \tag{41}$$

$$l \in V_{src} \tag{42}$$

$$\nexists v \in Path(l) : v \in V_{san} \wedge Suff(v, dfp^l_{AC}) \tag{43}$$

A vulnerability exists if tainted data from a leaf l (source) reaches the root $root_{AC}$ (sink) without passing any sanitization function from set V_{san}. For each vulnerability type, the source, sanitization functions and sinks are defined. The different sanitization functions defined in V_{san} are sufficient to prevent a vulnerability depending on the vulnerability type. For example, a sanitization function to prevent SQL Injection is usually not sufficient to prevent XSS attacks. The code example is not vulnerable because two sufficient sanitization methods for XSS are used.

5.2. Possible Injection Path

A possible injection path (PIP) is a data flow path in the source code that can be transformed into a vulnerability. The sets of sources and sinks are extended by additional sources and sinks that are usually secure to use. We define the set of PIP sources as $P_{src} \supseteq V_{src}$ and the set of PIP sinks as $P_{dst} \supseteq V_{dst}$. As an example, a secure source would be a function that only returns an integer value from the user. A secure sink could, e.g., be a bind query function from a parameterized SQL query.

Definition 6. The set Pip_{AC} contains possible injection paths. A possible injection path pip^l_{AC} is a data flow path dfp^l_{AC} with the following properties:

$$root_{AC} \in P_{dst} \tag{44}$$

$$l \in P_{src} \tag{45}$$

$$\exists v \in Path(l) : v \in V_{san} \wedge Suff(v, dfp^l_{AC}) \tag{46}$$

$$\vee l \notin V_{src} \vee root_{AC} \notin V_{dst} \tag{47}$$

The PIP definition is similar to a vulnerability definition. A PIP exists if data from a leaf node $l \in P_{src}$ (source) reaches the root node $root_{AC} \in P_{dst}$ (sink). At least one sanitization function ($p \in V_{san}$) in the path ($Path(l)$) has to be found or at least one secure source ($l \notin V_{src}$) or secure sink ($root_{AC} \notin V_{dst}$) has to be found. These requirements ensure that a PIP is not already a vulnerability. The nodes of the path from l to $root_{AC}$ are contained in $Path(l)$ and represent the data flow. That path will be used to transform the vulnerability with different source code patterns.

The code example is by definition a PIP. The path from *_GET* variable leaf $l \in P_{src}$ reaches the root node that represents the sink echo ($root_{AC} \in P_{dst}$). As requirements, the sanitization functions *intval*() and *isNumeric*() are found. The next section explains the required code changes to perform insecurity refactoring.

5.3. Injecting a vulnerability

The transformation of a PIP into a vulnerability uses the following transformation sets:

$$T_{src} \subseteq (P_{src} \times V_{src}) \tag{48}$$

$$T_{dst} \subseteq (P_{dst} \times V_{dst}) \tag{49}$$

$$T_{san} \subseteq (V_{san} \times V_{san}) \tag{50}$$

For the source transformation set T_{src}, secure source functions $p_{src} \in P_{src}$ are mapped to insecure functions $v_{src} \in V_{src}$. In the same manner, secure sinks $p_{dst} \in P_{dst}$ are mapped to insecure sinks $v_{src} \in V_{src}$. The sanitization functions are mapped to each other and depending on the $Suff$ function that can be used to make the sanitization functions insufficient resulting in vulnerabilities. The sets T_{src}, T_{dst} and T_{san} only map functions that can be replaced with each other without breaking the insecurity refactoring definition.

The possible injection path pip^l_{AC} can be transformed into a vulnerability $vuln^l_{AC}$, if the following **condition** check holds:

$$Check : pip^l_{AC} \to Boolean \tag{51}$$

$$Check(pip^l_{AC}) = \bigwedge_{v \in Path(l)} \begin{cases} Ch_{src}(v), & \text{if } v = l \\ Ch_{dst}(v), & \text{if } v = root_{AC} \\ Ch_{san}(v), & \text{if } v \in V_{san} \\ True, & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} Ch_{src}(l) &= l \in V_{src} \vee \exists(l, l') \in T_{src} \\ Ch_{dst}(r) &= r \in V_{dst} \vee \exists(r, r') \in T_{dst} \\ Ch_{san}(v) &= \neg Suff(v, pip^l_{AC}) \\ &\quad \vee \exists(v, v') \in T_{san} \wedge \neg Suff(v', pip^l_{AC}) \end{aligned}$$

The PIP condition check checks that for all ($\bigwedge$) nodes if it is a sufficient sanitization, secure sink or secure source that there exists an insecure representation. That ensures that the PIP can be transformed into a vulnerability. If this condition check returns *False* for a PIP pip^l_{AC}, it means that additional transformations in the transformations set are required to inject a vulnerability.

The transformation of a PIP into a vulnerability requires to modify the ACID tree. The replacement function is defined as follows:

$$\begin{aligned} Replace^{v'}_{v}(T_{AC}) &= T'_{AC} \\ &= (V'_{AC}, E'_{AC}, \lambda'_{AC}, \mu'_{AC}) \end{aligned} \tag{52}$$

where

$$\begin{aligned} V'_{AC} &= V_{AC} \setminus \{v\} \cup \{v'\} \\ E'_{AC} &= E_{AC} \setminus \{(v, u)\} \cup \{(v', u)\} \\ &\quad \setminus \{(w, v)\} \cup \{(w, v')\} \\ (v, u) &\in E_{AC} \\ (w, v) &\in E_{AC} \end{aligned}$$

The *Replace* function replaces a node v in an ACID tree T_{AC} with the node v'. It requires to connect the old edges that point to and from v to the replaced node v'.

Definition 7. A possible injection path pip^l_{AC} that passes the condition check can be transformed into a vulnerability $vuln^l_{AC}$ with the following function:

$$Tf : Pip_{AC} \to Vuln_{AC} \tag{53}$$

$$\begin{aligned} Tf(pip^l_{AC}) &= (G'_{AC}, l') \\ l' &= Tf_{src}(l) \\ G'_{AC} &= (Replace^{Tf_{src}(l)}_{l} \circ Replace^{Tf_{dst}(root_{AC})}_{root_{AC}} \\ &\quad \bigcirc_{v \in Path(l)} Replace^{Tf_{san}(v)}_{v})(G_{AC}) \end{aligned}$$

where

$$Tf_{dst}(r) = \begin{cases} r, & \text{if } r \in V_{dst} \\ r' : (r, r') \in T_{dst}, & \text{otherwise} \end{cases}$$

$$Tf_{src}(l) = \begin{cases} l, & \text{if } l \in V_{src} \\ l' : (l, l') \in T_{src}, & \text{otherwise} \end{cases}$$

$$Tf_{san}(v) = \begin{cases} v, & \text{if } v \notin V_{san} \\ v, & \text{if } \neg Suff(v, pip^l_{AC}) \\ v' : (v, v') \in T_{san} \wedge & \neg Suff(v', pip^l_{AC}), \text{ otherwise} \end{cases}$$

The ring operator ($\circ$) represent that all those functions have to be processed to transform a PIP into a vulnerability. In words, the transformation is done by replacing a secure sink ($l \notin V_{dst}$) by an insecure sink ($l \in V_{dst}$). In the same manner, a secure source is replaced by an insecure source. Additionally, all the sanitization functions in the path from source to sink have to be replaced by insufficient sanitization methods. In the code example, the functions *isNumeric* and *intval* ($\in V_{san}$) have to be replaced by insufficient sanitization methods ($\notin V_{san}$).

6. Implementation

This section describes the implemented PL/V pattern language that is used to detect and inject source code patterns to transform a PIP into a vulnerability.

6.1. The PL/V pattern language

Source code patterns are described in the PL/V language. PL/V is a context-free language that can be described in BNF as shown in Figure 10.

A *Pattern* consists of multiple code lines. If it is only a single code line, it can be an expression or a statement. Multiple code lines represent a statement list in which each line must be a statement. A code line has an identifier *id* and a parameter list *PrmList*. The identifier represents a language pattern. Language patterns are used to decouple the source code patterns from specific programming languages. For example, the pattern <=> (*%in*, *%out*) represents an assignment. It uses the symbol = as an identifier. Accordingly, a language pattern exists for the id = . The language patterns contain the information on how the Abstract Syntax Tree representation of a specific language pattern looks like. This allows to generate the Abstract Syntax Tree of the language patterns.

```
<Pattern>   ::= <CodeLines>
<CodeLines> ::= <CodeLine> "\n" <CodeLines>
              | <CodeLine>
<CodeLine>  ::= "<" <id> ">" <PrmList> ")"
<PrmList>   ::= <Param> "," <PrmList>
              | <Param>
<Param>     ::= <Pattern>
              | "%var" | "%in" | "%out"
              | literal
              | <Any>
<Any>       ::= "<any>" | "<any>..." | "<any>?"
              | "<any>?..."
```

Fig. 10. BNF of PL/V language.

The parameter list can contain other patterns, literals, variables and any nodes. The variables input (*%in*) and output (*%out*) are special case variables that can be used to chain source code patterns. Additionally, this allows to get input or output nodes of source code patterns by using the $ASTNode_x$ function.

In the example, the input is the expression that represents the value of the assignment. The output is the variable that will be assigned to.

Source code patterns can contain $< any >$ parameters with optional ("?") and multiple suffixes. An "any" parameter means that the parameter can be anything. The optional suffix ("?") specifies that the parameter does not have to exist. The "multiple" suffix ("...") specifies that any number of parameters can occur, but at least one parameter has to exist. A combination means that any number of parameters can occur including none. Literals are fixed values that are used in the corresponding pattern.

All patterns must contain a input *%in* and output *%out* variable except of a source and sink pattern. A pattern representing a source has only the fixed output variable. In contrast, a pattern of a sink only has the fixed input variable. Other variables can be used inside the pattern. For example, if a pattern requires accessing a specific key of an array, the key can be set as a variable *%var* and be used in further patterns.

The following example stems from our patterns Insecurity Refactoring (2022). It defines the sanitization pattern representing *htmlspecialchars*:

$$< call > (htmlspecialchars, \%in, < any > ())$$

The pattern uses the *call* language pattern. The *call* pattern requires a literal (*htmlspecialchars*) to define the function name in the Abstract Syntax Tree. The *%in* defines the input parameter. The *htmlspecialchars* function has an optional parameter that is represented as $< any >$. The output is the return value of the function.

For ACID tree construction, different source code patterns are required. For example, concatenations and the coding constructs are represented in the PL/V language. To find source code patterns it is required to have an equal check for each pattern. The following function allows to check if a part of the Abstract Syntax Tree from the Code Property Graph matches the pattern:

$$Match_p(v) = \begin{cases} False, & \text{if } \neg Type_p(v) \\ True, & \text{if } p = \emptyset \\ \bigwedge\limits_{p_i, v_i \in C_{pat}(p,v)} Match_{p_i}(v_i), & \text{otherwise} \end{cases} \tag{54}$$

The $Type_p(v)$ function checks if the type of the Abstract Syntax Tree node v equals the type of pattern node p. The $C_{pat}(p, v)$ function returns pairs of the children from the AST node v and pattern node p. $Match_p(v)$ checks recursively if the pattern node type is the same as the node type from the Abstract Syntax Tree. Parameter v represents the root node of the Abstract Syntax Tree that is checked and p is the root node of the pattern Abstract Syntax Tree. For simplicity, the *any* nodes are not specified in the $Match_p$ function. It simply checks based on the suffixes if the corresponding parameters exist or not.

The different variable nodes (*%in*, *%out*, *%var*) are used to represent important nodes. They are defined in the PL/V language. The following function allows to get the corresponding node in the Abstract Syntax Tree G_A based on the variable node x:

$$ASTNode_x(p, v) = \begin{cases} v, & \text{if } p = x \\ \bigcup\limits_{p_i, v_i \in C_{pat}(p,v)} ASTNode_x(p_i, v_i), & \text{otherwise} \end{cases} \tag{55}$$

It is a recursive function that searches for the same position in the Abstract Syntax Tree of the Code Property Graph starting from node v as in the sub tree of the pattern starting from node p.

6.2. Context analysis

The ACID tree is an analysis model that is used to evaluate the context. For each data flow path dfp^l_{AC} the context can be specified. A context c can be identified by what is concatenated before the input (*pre*) and what is concatenated after the input (*post*). Instead of formalizing the context check, we define for each context ($c \in C$) the function $IsContext_c(pre, post)$. The inputs *pre* and *post* are both string values. The function returns a boolean value and uses different checks to specify if the context c exists for the inputs.

We define the following function to get a set of all contexts for a data flow path dfp^l_{AC}:

$$Context(dfp^l_{AC}) = \{c \in C_{ctx} \text{ and } IsContext_c(Up_{pre}(l), Up_{post}(l))\} \tag{56}$$

It uses the recursive functions $Up_{pre}(l)$ and Up_{post} to get the string values that the input is concatenated with. These functions are defined as follows:

$$Up_{pre}(v) = \begin{cases} \bigcup\limits_{c \in Bef(v)} Down(p(c)), & \text{if } p(v) = \wedge \\ Up_{pre}(p(c)), & \text{otherwise} \end{cases} \tag{57}$$

$$Up_{post}(v) = \begin{cases} \bigcup\limits_{c \in Aft(v)} Down(p(c)), & \text{if } p(v) = \wedge \\ Up_{post}(p(c)), & \text{otherwise} \end{cases} \tag{58}$$

Up is a recursive function that iterates from the leaf upwards to the root node. If the parent of node v is a concatenation ($\wedge$), data will be concatenated to the input data. Accordingly, that concatenated data is the context of the input data. Based on *pre* or *post* context, the corresponding siblings before or after of the input nodes are analyzed using the *Down* function. In the initial code example, on line 15 there is a concatenation of the variable *$name* and the variable *$page*. The variable *$page* is the input from the user and the variable *$name* is the context that is concatenated.

The function

$$Down(v) = \begin{cases} String(v) & \text{if } Type_{string}(v) \\ \bigcup\limits_{c \in C(v)} Down(c), & \text{if } v = \wedge \\ Down(First(v)), & \text{if } v = \oplus \\ Down(C(v)), & \text{otherwise} \end{cases} \tag{59}$$

is a recursive function that iterates the tree downwards. If a concatenation is found, the recursive function of the children nodes will be united. A problem may occur if an excluding node $\oplus$ is reached. In the example on line 6, the children from

Parameter(\$*name*) are excluding. The downwards recursive function has to decide which mutually excluding child will be used for the context analysis. Different approaches can be used. Either one child is selected and used for the context analysis or all children are checked to see if they result in a similar context. We decided to use the context of the first child. It is a simple heuristic under the assumption that the context will not differ from other sub trees. Even if the context is different based on the different sub trees, at least one sub tree has the analyzed context. Accordingly, the approach can only ensure the chosen case will be exploitable.

The context analysis of the ACID tree sample in Figure 9 shows that the user-provided data is concatenated with the String *news* and <*a href*='www.url.com/ as pre context. The post context is the String ' >*link* </*a*>. Accordingly, the output on the web page will be:

<*a href*='www.url.com/news*[input]*' >*link* </*a*>.

In the code example, there is a potential Cross Site Scripting sink. Accordingly the Cross Site Scripting relevant context checks are required. In the example, the context check for HTML attribute context and *inside apostrophes* context will return true.

6.3. Insert data flow pattern

A main goal of insecurity refactoring is to create learning examples. Previous research Schuckert et al. (2017) Schuckert et al. (2018) showed that many interesting source code patterns are data flow source code patterns. The path $Path(l)$ defines the nodes from a source to a sink. It also represents the data flow of the PIP. Depending on what kind of learning example should be created, different data flow patterns are interesting. For example, some data flow patterns are difficult to detect by static code analysis tools Schuckert et al. (2019). If the learning examples should be more focused on *Capture the flag* (CTF) events, data flow patterns can be added that, for example, teach specific techniques like dynamic function calls. Also data flow patterns can be used to make the vulnerability difficult to detect by dynamic analysis tools (e.g. fuzzers).

Definition 8. The transformation of data flow patterns is defined as a tuple:

$$T_{df} = (D, M, \mu_R) \tag{60}$$

The set D defines all data flow patterns. The set $M \subseteq (D \times D)$ defines the patterns that can be replaced with other data flow patterns. The function $\mu_R : D \to R$ defines what requirements $r \in R$ are required for the inserted data flow pattern $d \in D$. The set R contains all requirements. A requirement $r \in R$ is a combination of a context $c \in C_{ctx}$ and a boolean that defines if the context must exist or must not exist. If all requirements for a data flow pattern are fulfilled, the data flow pattern can be injected without breaking the insecurity refactoring definition.

Usually the patterns that used to be replace with interesting data flow patterns are simple like an assignment. Interesting patterns represent different difficulties of the vulnerabilities. The requirements $r \in R$ have to be fulfilled to transform the source code maintaining the insecurity refactoring requirements. For example, one data flow pattern will redirect to the main page if the tainted variable is not an integer. The pattern does not contain an exit statement and the source code later on is still executed (Pattern found in CVE-2013-3524). This pattern requires that the initial PIP contains a restriction to integer only variables. The requirement ensures that the program will still run as normal as long as only integer values are inserted. But it will change its external behavior as soon as attackers insert unintended values like a string. The patterns are searched in the path $Path(l)$ using the $Match_p(n)$ function.

```
<def_func>(sanitize, <param_list_1>(<$>(a)),
    <stmtlist>(<return>(<$>(a))))
<=>(<$>(func), <s>(sanitize))
<=>(%out, <call_v>(func, %in))
```

Fig. 11. Function call by string described in the PL/V language.

6.4. Source code modification example

All the code modifications are based on the transformation sets (T_{src}, T_{dst}, T_{san} and T_{df}). Each element of the sets can define different variables that are required to perform a code modification. The modifications are done on the Abstract Syntax Tree (G_A). The variables of the PL/V language represent sub trees of G_A. Figure 12a shows the code example and Figure 12b shows the insecurity-refactored source code. Applying the rules to inject a vulnerability requires to replace the *intval* function and *is_numeric* with an insufficient sanitization pattern $p \in T_{san}$. The source and sink do not require any modifications. As described previously, data flow patterns allow to introduce different source code patterns. In the example, the assignment ($<=> (\%out, \%in)$) pattern on line 7 is used to introduce a data flow pattern. For that assignment the output *%out* is the AST sub tree that represents the variable *page*. For the input *%in* the correspding AST sub tree represents the expression that is assigned to the variable. In that case, it is the function call *getParam*. Figure 11 shows the inserted data flow pattern.

This pattern is difficult for static code analysis tools Schuckert et al. (2019). The insecurity-refactored source of the PIP is shown in figure. On line 14, the sanitization function *is_numeric* is replaced by the insufficient sanitization function *is_string*. On line 15, the sanitization function *intval* is replaced by the *htmlspecialchars* sanitization function. The sanitization function is insufficient for the *inside apostrophes* context. It would require an *inside quotes* context to be sufficient. A potential Cross Site Scripting attack could inject a *onclick* parameter with Javascript payload. Lines 7 to 12 show the source code pattern that is difficult for static code analysis tools. It represents a dynamic function call using a string value.

All the modifications to inject a vulnerability are done by modifying the Abstract Syntax Tree. The next step is to revert the modified Abstract Syntax Tree back into actual source code. A simple program was written that generates PHP source code based on the Abstract Syntax Tree. An Abstract Syntax Tree uses some kind of abstraction to unify functions with the same functionality. For example, <*?*= and *echo* are both represented by an *echo* function call. The current approach checks the Abstract Syntax Tree to determine what lines of code are modified. Only the modified lines are replaced by the generated PHP source code and other lines of the files are maintained. This diminishes the chance that an abstraction breaks the source code.

The injection of vulnerabilities is semi-automated. For each PIP, the tool shows the critical sanitization functions that have to be replaced. The tool provides a list of sanitization functions that are insufficient for the corresponding context. In addition, patterns can be selected to be injected. After selecting the injected patterns, the tool checks if all sanitization functions will be replaced with insufficient sanitization functions. If all of them are selected, the vulnerability will be injected. For a fully automated approach, the patterns could be selected randomly.

7. Evaluation

The evaluation explores whether the Insecurity Refactoring approach is applicable to real projects. Additionally, it is important to see if the insecurity-refactored projects break the Insecurity Refac-

```php
<?php
function getParam($param){
  return $_GET[$param];
}

function page($debug, $name){
   $page=getParam('page');






   if(is_numeric($page)){
      $out = $name . intval($page);
      $out = "<a href='www.url.com/" .
           $out . "'> link </a>";
   }
   else {
      $out = "Unknown page";
   }

   echo $out;
}
?>
```

(a) The original code example.

```php
<?php
function getParam($param){
  return $_GET[$param];
}

function page($debug, $name){
   function sanitize($a)
   {
     return $a;
   }
   $func = "sanitize";
   $page = $func(getParam("page"));

   if(is_string($page)){
      $out = ($name . htmlspecialchars($page));
      $out = "<a href='www.url.com/" . $out . "'>
           link </a>";
   }
   else {
      $out = "Unknown page";
   }

   echo $out;
}
?>
```

(b) Insecurity-refactored code example.

Fig. 12. Insecurity refactoring using a data flow pattern that is difficult for static code analysis tools (function call by string).

Table 1
Possible injection path data set.

Type	Sources (P_{src})	Sanitization (V_{san})	Passthrough (V_{pass})	Sinks (P_{dst})
All	9	98	164	•
XSS	•	•	•	9
SQLi	•	•	•	77
Eval	•	•	•	1
Unserialize	•	•	•	1

toring definitions. The main condition is that the projects can still be executed for normal use. Also the usage as learning example is evaluated.

7.1. Open source projects

We developed a tool to perform insecurity refactoring on PHP projects Insecurity Refactoring (2022). First of all, we want to see if insecurity refactoring can be used to inject vulnerabilities in open source projects. This requires to define the set of sources, sinks, etc. Table 1 shows how many entries are in each set. We retrieved the sets by reviewing the PHP documentation PHP Documentation (2021). Each sanitization function from V_{san} that passes through data is also found in the passthrough data set (V_{pass}). The other passthrough functions are mainly functions to manipulate string values. The SQLi sinks contain different functions because each database has different PHP drivers. We added all functions from SQL database drivers we found in the PHP documentation. Only 9 sources are in our data set that are mainly functionality from PHP like the global array *_GET*. *Eval* and *unserialize* are different vulnerability types. Each of them is represented as a sink for their category.

A crawler tool was written that crawls GitHub Github (2022) for projects that contain PHP source code. The corresponding source code is then checked for PIPs. Table 2 shows the results. 307 open source projects were scanned. In 25 of these projects PIPs were found. Accordingly, the tool could inject vulnerabilities in 8.1% of the projects. It also shows that most of the PIPs are related to

Table 2
Possible injection paths found in 25 open source projects out of 307 scanned projects.

	PIP	True Positive (Vuln)	False Positive (Vuln)
XSS	221	16	57
SQLi	98	37	10
Eval	1	0	1
Unserialize	3	2	0
	323	55	68

Cross Site Scripting. Not many projects contained PIPs related to textiteval or textitunserialize. We also found several vulnerabilities. Those reports were reviewed: 55 true vulnerabilities and 68 false positive reports. Most of the vulnerabilities were found in installation and testing files or were deliberate vulnerabilities. Three vulnerabilities were potentially dangerous vulnerabilities in commonly used projects. These vulnerabilities were reported by us to developers and they confirmed and patched the vulnerabilities (CVE-2020-27163, CVE-2021-3318, CVE-2021-26716). The vulnerability reports also included 68 false positive reports. In these cases pre-conditions prevent an exploitation. These pre-conditions were outside the ACID tree and could not be detected. The false positive vulnerability reports show that not all sanitization approaches can be detected. Nevertheless, PIPs that are used for insecurity refactoring have to contain a detected sanitization function to inject a vulnerability. This decreases the possibility that an injected vulnerability is not exploitable. One problem for Cross Site Scripting ex-

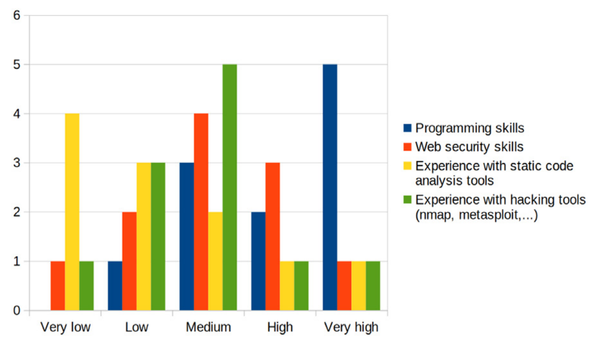


Fig. 13. Pre-Survey CTF event to check the skill level. (n=11)

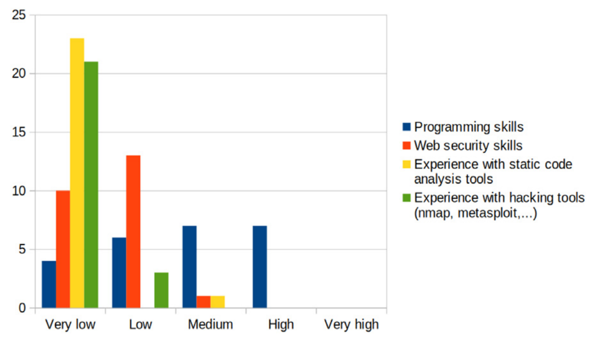


Fig. 14. Pre-Survey student exercise event to check the skill level. (n=24)

ists if the *Content-Type* is set to a secure type (e.g. *plain/text*). The ACID tree does not contain the *Content-Type*. If a PIP is found in a file that sets a secure *Content-Type*, the injected vulnerability will not be exploitable. If no sanitization function is found, but a special sanitization function exists, it will not be selected as as PIP. Accordingly, those false positive reports do not impact the possibility for insecurity refactoring.

The results demonstrate that the concept of insecurity refactoring works on open source projects. By increasing the data set of sources and sinks, the chances of finding PIPs can be increased. Adding support for object-oriented data flows to the initial Code Property Graph will increase the findings.

7.2. Learning examples

Initial evaluation shows that PIPs are found in open source projects. The next step is to see if the refactoring itself works without breaking the functionality of the projects. Additionally, it is important to check if the injected vulnerabilities can actually be used to teach software security. For the evaluation, two exercises were arranged for two different groups. The exercises are projects Insecurity Refactoring code samples (2022) that were insecurity-refactored to contain different vulnerabilities. The groups and the corresponding exercises are described in the following sections.

Surveys were used for evaluation of the experiment. At the beginning, a pre-survey was provided to get information about the skill level of attendees. The exercise itself was a bit different for each group. Both used insecurity-refactored projects. For both groups the insecurity-refactored projects were hosted in virtual machines. This allowed to check if the insecurity refactoring actually maintained the external behavior of the programs. After the experiment, a post-survey was provided to see if the exercises were perceived as difficult or realistic and if attendees experienced a skill increase.

7.2.1. Experienced group

The first group was a mix of people with different backgrounds. All of them were training for an upcoming cyber security challenge. Figure 13 shows the skill level in different categories. The skill level in programming is overall very high. Also the web security skills are towards medium high rating. The experience with different hacking tools is seen as medium and the experience with static code analysis is low. This group has already experience with capture the flag events from attending other training events. Further, the group is described as the experienced group ("Exp.").

Based on the experience of the group, the idea for this exercise was to provide the attendees with the insecurity-refactored projects in virtual machines like in a CTF event. The attendees were supposed to use their own strategies for detecting vulnerabilities. Attendees had access to the virtual machines that also allowed them to access the source code of the projects.

Table 3 shows an overview of the insecurity-refactored projects. The following four projects were used for insecurity refactoring: *phpBB, EmonCMS, phpRedisAdmin* and *Adminer*. Overall we injected four SQL Injection vulnerabilities and three Cross Site Scripting vulnerabilities spread over the four projects. The phpBB project uses self defined functions for getting user data. We have added a project specific pattern that represents that functions. Without that pattern the Insecurity Refactoring tool would not be possible to detect PIPs in phpBB. Accordingly, the phpBB did not add any PIPs to the initial evaluation based on scanning open source projects. The project specific pattern can be found on GitHub but is disabled at default. Data flow patterns were added, of which three are difficult for static code analysis tools and two difficult for dynamic testing tools. One pattern also used the *function call by string* pattern described earlier. One vulnerability was a plain SQL Injection without any sanitization methods and two vulnerabilities used an insufficient sanitization method for the vulnerability or context. The goal was to create vulnerabilities with different difficulty levels. Additional difficulties can be achieved by adding data flow patterns. Table 3 shows these patterns and the corresponding difficulties for different approaches are listed. The difficulties vary for the different approaches to discover a vulnerability. Static code analysis tools (sca), dynamic testing tools (dyn) and manual inspections (man) have different difficulty levels. For example, a dynamic tool has problems to detect backdoors that require specific inputs to bypass sanitization. In contrast, static code analysis tools usually have more problems detecting different dynamic programming approaches or specific source code patterns Schuckert et al. (2019) Schuckert et al. (2020).

Attendees were grouped into four teams who worked together to find the vulnerabilities deployed in the virtual machines. The groups had to provide a report to score points. The report had to contain how the students discovered the vulnerability, how the vulnerability can be exploited and how the vulnerability can be patched. Having a report about discovery, exploitation and patching allowed us to analyze how the teams solved the tasks. The event ran for 24 hours. In the first hours, the groups had to discover the vulnerabilities without any further help. After the initial 10 hours, hints about the vulnerabilities were released. For example, "Some users reported that changing the style of phpBB is buggy." In this case the injected vulnerability used the style parameter in *user.php* for a SQL Injection vulnerability.

7.2.2. Beginners group

A second evaluation as a learning example was done with a group of students. The students were relatively new to software security. Figure 14 shows their initial skill levels. It shows that the programming skills are higher than the web security skills. This kind of skill level was expected because the students study computer science and the exercise was done for a software security class. The group will be described as beginners group ("Beg.").

Table 3
Insecurity-refactored projects for the experienced group (Exp.).

Project	Type	Input parameter	DF Pattern	Insuff. San.	special	G1	G2	G3	G4
phpBB	SQLi	user.php (style)			hint	✓	✓	✓	✓
phpBB	SQLi	memberlist.php (g)	redirect [man.]						
phpBB	SQLi	posting.php (t)	backdoor int cast [dyn.]						
emonCMS	XSS	compare.php (feedA)	Class storage [sca]	htmlspecialchars		✓	✓	✓	
emonCMS	SQLi	admin_controller.php (perPage)		htmlspecialchars				✓	
phpRedisAdmin	XSS	view.php (page)	backdoor expl/imp. [dyn/sca]		.git dir		✓	✓	
Adminer	XSS	table.inc.php (table)	function call by string [sca]						

Table 4
Insecurity-refactored projects for the beginner group ("Beg.").

Project	Type	Input parameter	DF Pattern	Insuff. San.	special
phpBB	SQLi	user.php (style)			Parameter list
phpBB	SQLi	memberlist.php (g)	backdoor int cast [dyn.]		Parameter list
phpBB	SQLi	posting.php (p)	redirect [man.]		Parameter list
emonCMS	XSS	compare.php (feedA)	Deactivated default san. [sca]		Parameter list
emonCMS	SQLi	admin_controller.php (perPage)	function call by string [sca]		Parameter list
emonCMS	XSS	dailyhistogram.php (kwhd)	comparing different types [man]		Parameter list

Because the group was not experienced with using any static code analysis or dynamic testing tools, the exercise itself had to be different. Table 4 shows the insecurity-refactored projects that were used in this exercise. Only phpBB and EmonCMS were used to injected different vulnerabilities. This time four SQL Injection and two Cross Site Scripting vulnerabilities were injected in the projects. All of them contained different data flow patterns to again make it difficult for static code analysis tools or dynamic testing tools. Except for the very simple SQLi vulnerability in phpBB, no other vulnerabilities are the same as in the data set for the experienced group. Some inputs are the same, but different data flow patterns make them different from each other.

Because the students were not familiar with using static code analysis and dynamic tools, they got tutorials on how to use such tools. Also the students were provided with a static code analysis tool and a dynamic tool that they could use. For the exercise, the students had to scan the provided source code with the static code analysis tool. As the next step, they had to scan the insecurity-refactored projects with the provided dynamic tool. The students got the insecurity-refactored projects deployed in virtual machines and they separately got the corresponding source code. As the last step the students were provided with a list of the vulnerable parameters. This allowed them to check the tools' results and they could manually inspect the remaining undetected vulnerabilities. For each of the steps, the students had to report if it is possible to exploit the discovered vulnerabilities. No patching of the vulnerabilities was required. The time frame for this experiment was four weeks.

7.2.3. Results

First of all, the insecurity-refactored projects with different patterns were deployed in virtual machines. No strange behavior of the projects was reported. Accordingly, the insecurity-refactored projects performed normally as long as no vulnerability was exploited. One problem in the experienced group was revealed that at some point attendees found out that the latest version on GitHub had been used as the base for insecurity refactoring. Then they started to use the *diff* command on the insecurity-refactored projects to find further vulnerabilities. Table 3 shows for each group (G1-G4) what vulnerabilities have been reported. Three of seven vulnerabilities were not reported at all. A problem in the beginner group was that it was forced to do the exercise from home. Therefore, they could not be guided well to use the given tools and had to rely on the provided tutorials. This was a hurdle that many of the beginners could not overcome and not all of them finished.

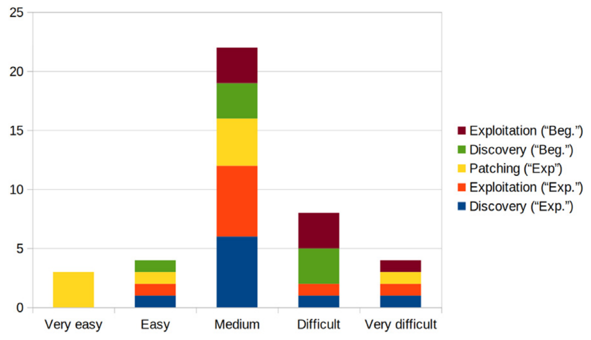


Fig. 15. Survey results on the difficulty of tasks. (Discovery, Exploitation and Patching) (n=16)

It was not mandatory to finish the exercise and only 7 of the initial 24 attendees actually finished the exercise.

In the post-survey, attendees were asked how difficult the tasks had been for them. Figure 15 shows the results combining the difficulty of discovery, exploitation and patching. For the beginners group the question about patching did not exist because they did not patch the vulnerabilities in their exercise. Overall, the results are towards medium difficulty with being a bit more towards the difficult side. For an exercise, the medium difficulty is optimal. It does not overwhelm learners and is not too easy to solve. Experienced attendees point out that they were a bit overwhelmed by the large projects. Additionally, they pointed out that the tasks were not isolated like in other CTF events. No such complaints were voiced in the beginners group, probably because they got a list of vulnerable parameters. The patching by the experienced group was described as more on the easy side. The reason is that most teams used the master version on GitHub as a patch solution. That is a correct solution, but does not require any skills to patch the vulnerability.

Figure 16 shows the results of the questions on how real the tasks were considering bug bounty or code inspections tasks in real life. The results of both groups are shown in the diagram. It indicates that the exercise was close to a real life example. Only two attendees answered that closeness to reality was low.

Attendees were asked how they think their software security skills improved by that event. The results are shown in Figure 17. First of all, two members of the experienced group did not see any skill increase by the event. The other 14 attendees answered that

Table 5
Comparing methods of LAVA, EvilCoder and Insecurity Refactoring.

Method	LAVA	EvilCoder	Insecurity Refactoring
Language	C/C+	C/C+	PHP
Detection	DUA (dynamic)	Code Property Graph 1. backward 2. forward (CFG)	ACID Tree Context of input
Injection	Use DUA in sink	Invalidate security mechanisms or use security anti patterns	1. Ins. sanitization function 2. Add data flow pattern
Realismn	Synthetic	Artificial	Patterns stem from CVEs

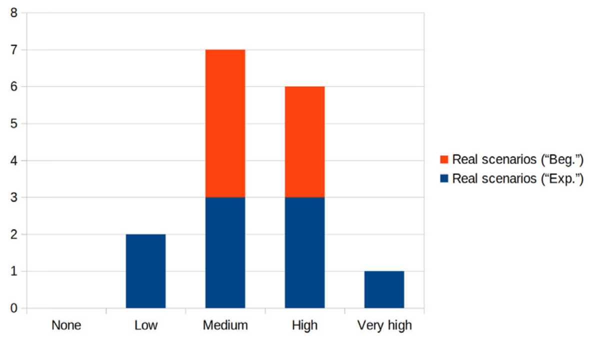


Fig. 16. Post-survey results about how similar the exercise is to real penetration testing/bug bounty scenarios. (n=16)

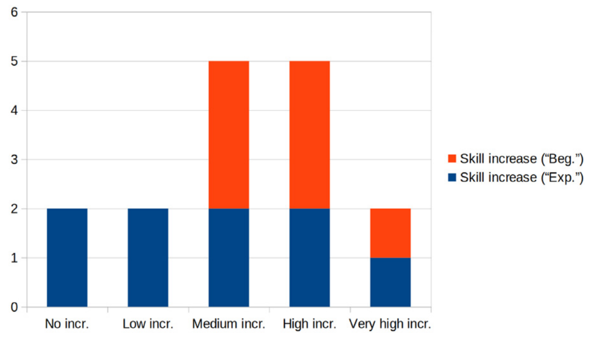


Fig. 17. Post-survey results if attendees experienced a skill increase from the exercise. (n=16)

they experienced a skill increase in software security skills by the event.

7.3. Comparative analysis

Previous approaches have used similar procedures to inject vulnerabilities. Table 5 shows an overview of methods to inject vulnerabilities from both LAVA and EvilCoder as well as Insecurity Refactoring.

7.3.1. Functional comparison

LAVA uses a dynamic taint analysis to detect DUAs (Dead, Uncomplicated and Available Data). In words, a DUA is a user-controlled input that does not change any control flows and is not concatenated with other variables. Then they search for attack points (ATP) which are near DUAs. An ATP is a sink that can be transformed to create a vulnerability. The vulnerability injection transforms the ATP by adding a conditional usage of the corresponding DUA.

Compared to our course of action, the dynamic approach requires a running setup of the program. First of all, that makes the scanning effort more difficult. Nevertheless, the detection should be more precise. Additionally, the authors state that the injected vulnerabilities are synthetic, therefore only exploitable if specific inputs are provided. The condition allows the program to run as intended as long as the specific input is not provided. Therefore, they state that it makes the vulnerabilities more realistic. Nevertheless, the injected vulnerability patterns do not stem from real vulnerabilities.

The EvilCoder approach uses a Code Property Graph. The detection of potential injection locations is done in two steps. In the first step, for all potential sinks a backwards taint analysis on the Code Property Graph is started to find sources. In the second step, for each potential path from a sink to a source, a forward analysis (source to sink) on the Control Flow Graph is started. In the forward analysis, it is searched for security checks that influence the control flow based on data from the tainted variable. These security checks are transformed to inject the vulnerability. It can either be injected by invalidating the security checks or by the use of a security anti-pattern. A security check is invalidated by transforming the conditions to always being true or false. A security anti-pattern transforms the sink to use patterns that are always critical. For example, a `printf("%s", buf)` is replaced by `printf(buf)`.

Compared to our approach, the injected vulnerabilities are artificial. The approach does not ensure the normal behavior of the program afterwards. The injected vulnerability might be triggered all the time. Their approach uses a concept to remove security checks. Many C/C++ vulnerabilities are related to memory bugs that make length checks critical. Our approach can replace security checks and functions that transform data (e.g. *htmlspecialchars()*). The approach to add anti-patterns is similar to our approach of replacing secure sinks with insecure sinks.

Overall, our approach is focused towards PHP and corresponding typical vulnerabilities. As PHP is typically used in web-based applications, the vulnerabilities heavily depend on the context. The ACID tree is another analysis model on top of the Code Property Graph that allows to analyze the context of given user input. This gives us the opportunity to be more specific whether a sanitization function is sufficient or not. The other approaches do not consider the context. LAVA tries to minimize that problem by using variables that are not concatenated with other variables. EvilCoder instead invalidates the whole security check independent of the context. In contrast, our approach is precise, which has the disadvantage of not finding as many potential injection paths. But it maintains the normal usage of the program. Our approach provides a PL/V pattern language that allows to describe the critical patterns. Additionally, our patterns stem from existing CVEs to maintain patterns of realistic vulnerabilities. By definition all injected vulnerabilities are artificial, including our approach. Insecurity Refactoring injects patterns that stem from CVE reports in existing projects to keep the vulnerabilities as realistic as possible. In addition, the data flow patterns can be used to introduce difficulties based on the pattern.

7.3.2. Experimental comparison

The EvilCoder approach to find PIPs is similar to our approach. An experiment with the same programs as input is not possible because EvilCoder uses C and our approach uses PHP as input. Nevertheless, we compare their results from scanning open source projects to our results from scanning open source projects in detail. Table 6 shows the results that EvilCoder got on four open source

Table 6
Comparing results of EvilCoder and Insecurity Refactoring.

	EvilCoder			
	libpng	vsftpd	wget	busybox
Lines of code	40,004	20,046	137,234	265,887
Sources	9	3	21	152
Sinks	98	13	453	573
Unique Source-Sink	158	22	22	30
Source-Sink paths	22,516	786	1,882	2,905
	Insecurity Refactoring			
	Adminer	EmonCMS	phpBB	phpRedisAdmin
Lines of code	27,606	26,383	289,800	2,022
Sources	752	138	552 (223)	210
Sinks	1,386	3,417	3,795 (3,795)	478
Unique Source-Sink	39	13	188 (0)	25
Source-Sink paths (PIPs)	65	14	292 (0)	30

projects and that we achieved for the same number of projects. The results include a special pattern for the custom *phpBB* function to retrieve user data. Without that pattern, Insecurity Refactoring cannot find a PIP. The results without the pattern are shown in brackets. First of all, the results show that Insecurity Refactoring finds a lot more sources and sinks compared to EvilCoder. A reason for that is that PHP web vulnerabilities have a different kind of sinks and sources. For example, for XSS every function that prints text on a web page will be a possible sink. This includes functions that just print static text. A unique source-sink stands for at least one data flow path between a specific source and sink. The Insecurity Refactoring flags an ACID tree that contains at least one path from source to sink as a PIP. It does not count each leave as an additional PIP. In contrast, source-sink paths count all possible paths that are found between sources and sinks. First of all, the results show that EvilCoder finds more unique source-sinks per given sources and sinks compared to Insecurity Refactoring. Compared to the lines of code, the unique source-sink pairs found are in a similar range. For the source-sink paths, EvilCoder finds more paths. The different code base and vulnerabilities might explain that. Nevertheless, an implementation difference here is that EvilCoder tracks each control flow path that can be taken. The ACID tree combines such control flow paths. Another path (split in the ACID tree) would only be created when an *if* statement contains a union that then will be represented as an excluding ($\oplus$) split.

As a next step, the vulnerability injection can be evaluated. EvilCoder ships only two kinds of instrumentation on the GitHub project. They state that it can be extended to create more variations. Here is a gap between their approach and ours. Our approach evaluates if a sanitization function is sufficient for a given context. EvilCoder only provides the possibility to replace an *if* statement with an instrumentation. Our approach allows more variations for a given PIP (source-sink). For example, for a given PIP it is allowed to replace a source with 5 other sources, 10 different data flow patterns can be inserted, and 9 different sanitization functions would be insufficient. This allows to inject a vulnerability in $5 * 10 * 9 = 450$ different permutations. This is an advantage of our approach over the EvilCoder approach. Small patterns can be defined and those patterns can be combined to inject vulnerabilities.

Nevertheless, the comparison between two tools that work with different vulnerabilities and on source code in different programming languages cannot be compared empirically. Our experiment shows that EvilCoder provides more possibilities to inject vulnerabilities that use different data flow paths. In contrast, the Insecurity Refactoring approach allows to inject many different permutations of a vulnerability.

8. Discussion

Insecurity refactoring is a novel method that injects vulnerabilities in projects based on source code patterns gathered from vulnerabilities in CVE reports. It shows that PIPs can be found and transformed into vulnerabilities. Also some patterns can be added that make detection by static code analysis tools difficult. One ethical question is if developing and publishing such a tool might be more harmful than useful. The main idea is to actually use insecurity refactoring to create learning examples. The tool could also be used maliciously to inject vulnerabilities in projects that are actually deployed in productive systems. For example, a malicious *Git* software could use insecurity refactoring to inject vulnerabilities before it pushes code changes to the *Git* server. Such an attack scenario requires that the *Git* server does not review pull requests. Another scenario might be that the *Git* client that pulls the source code is malicious. The client could perform insecurity refactoring on each pull request. This is a possible attack scenario but requires to add the malicious *Git* client on the server in the first place. We see such attack scenarios as more artificial than actually relevant in practice.

As learning examples, a defined difficulty of the vulnerabilities would be beneficial. Our evaluation shows that most of the attendees reported skill increase attending an event that used vulnerabilities generated by insecurity refactoring. Some form of difficulty rating for the different patterns would be useful. For now we can only predict the difficulty based on how large the initial project source code is and whether we added some special patterns. The results of the evaluation show that some attendees would like to have hints as to where vulnerabilities are. Accordingly, the scenario of the exercise itself is also important. Insecurity refactoring allows to inject vulnerabilities in real projects to get vulnerability examples as real as possible. Nevertheless, the exercises in which the insecurity-refactored projects are used may be very different. The results show in two different scenarios that the insecurity-refactored projects can be used as learning examples.

The difficulty varies based on what kind of learning example the insecurity-refactored source code is used for. If it is used to teach the use of static code analysis tools, a vulnerability without a special difficult static code analysis pattern is not difficult. But it might be difficult if the vulnerability has to be found manually. In the end, the difficulty of the insecurity-refactored vulnerabilities heavily depends on the task.

Our evaluation on open source projects showed a problem in finding control functions using the *Ctrl* function. This is a classical NP-hard problem because all possible paths from one statement to another statement have to be created and each of these paths has to be checked if it contains any sanitization check methods. Most of the time, possible paths are short and the query runs fast. Nevertheless, some projects contain so many possible paths (high complexity) that the query run time increases to an inconveniently long period (>2 minutes) on modern hardware. As a solution we scanned in two steps. The first step ignores any control function checks. If the first step finds a PIP or vulnerability, a second analysis is done using the control function checks.

Another problem is that the control property graph does not support object-oriented data flow. Method calls from objects are resolved correctly, but data that is stored in object variables is not tracked. This decreases the chance to find PIPs. Especially for SQLi, many database drivers are stored in objects (db wrapper) or the queries are constructed using data-represented objects Schuckert et al. (2017).

The evaluation shows positive results for a small survey size ($n = 9 + 7 = 16$). The results of the evaluation as a code inspection task shows that the small test group had skill improvements. The small survey size cannot be used to statistically proof that the insecurity refactored projects are always beneficial as learning examples. At least for that small test group it showed skill improvements. Future work should use larger test groups ($n > 100$) to statistically verify the initial results of the small sample size. However, the usefulness of software security exercises not only depends on the vulnerability itself. The results show that insecurity-refactored vulnerabilities are usable for software security exercises. One problem of such exercises is the time it takes to create vulnerabilities in a real scenario. The results show that insecurity refactoring can inject vulnerabilities with different patterns into existing projects. Accordingly, the scenario where vulnerabilities appear is as real as possible. Overall, the concept of insecurity refactoring works on open source projects without violating the definition of insecurity refactoring (not changing the external behavior in normal usage).

9. Conclusion

Our approach for insecurity refactoring shows that vulnerabilities can be injected into open source projects by using static code analysis approaches. The ACID tree is introduced as an analysis model for finding PIPs and vulnerabilities. Finding locations of PIPs has the same limitations as finding vulnerabilities in the first place. A precise approach was used to mitigate any false positive results where injected vulnerabilities would have a high chance of not being exploitable. A false positive PIP might break the normal use of the project, hence breaking the insecurity refactoring definition. The PIP can be simple. The injected vulnerability can be made difficult by adding data flow patterns. These patterns can be so difficult that the ACID tree approach cannot detect them anymore. Accordingly, the injection of vulnerabilities can be a lot easier than the detection of the injected vulnerabilities. If any useful attack scenarios of insecurity refactoring are found, the automated detection of these vulnerabilities will be more difficult.

The focus of insecurity refactoring is to inject vulnerabilities with different source code patterns. The PL/V pattern language allows to define the source code patterns in an independent language. To extend the tool to support other programming languages only the language patterns have to be rewritten and the Code Property Graph has to be created for that language. A first evaluation shows on a small sample size that insecurity refactoring can be used to teach software security skills. Compared to other approaches our focus relies on to create vulnerabilities realistic as possible. The approach shows that the concept works with different source code patterns. The different patterns also allow to create many permutations of vulnerabilities. This enables repeatedly using insecurity refactoring to teach software security skills.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Felix Schuckert: Conceptualization, Methodology, Software, Investigation, Writing – original draft, Visualization. **Basel Katt:** Conceptualization, Writing – review & editing, Supervision. **Hanno Langweg:** Conceptualization, Writing – review & editing.

Data availability

The data is available on GitHub.

References

Alhuzali, A., Gjomemo, R., Eshete, B., Venkatakrishnan, V.N., 2018. NAVEX: Precise and scalable exploit generation for dynamic web applications. Proceedings of the 27th USENIX Security Symposium 377–392.

Backes, M., Rieck, K., Skoruppa, M., Stock, B., Yamaguchi, F., 2017. Efficient and Flexible Discovery of PHP Application Vulnerabilities. Proceedings - 2nd IEEE European Symposium on Security and Privacy, EuroS and P 2017 334–349. doi:10.1109/EuroSP.2017.14.

Boland, T., Black, P.E., 2012. Juliet 1.1 C/C++ and Java Test Suite. Computer 45 (10), 88–90. doi:10.1109/MC.2012.345.

Burket, J., Chapman, P., Becker, T., Ganas, C., Brumley, D., 2015. Automatic problem generation for *Capture – the – Flag* competitions. 2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15).

Chapman, P., Burket, J., Brumley, D., 2014. {PicoCTF}: A {Game-Based} computer security competition for high school students. 2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14).

Deo, N., 1974. Graph Theory with Applications to Engineering and Computer Science (Prentice Hall Series in Automatic Computation). Prentice-Hall, Inc., USA.

Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., Whelan, R., 2016. LAVA: Large-Scale Automated Vulnerability Addition. Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016 110–121. doi:10.1109/SP.2016.15.

Du, W., 2011. SEED: Hands-on lab exercises for computer security education. IEEE Security and Privacy 9 (5), 70–73. doi:10.1109/MSP.2011.139.

Fowler, M., 1999. Refactoring: Improving the design of existing code. Addison-Wesley Professional.

Github, 2022. https://github.com/.

Insecurity Refactoring, 2022. https://github.com/fschuckert/insecurity-refactoring.

Insecurity Refactoring code samples, 2022. https://github.com/fschuckert/insec_samples.

Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M., 2018. Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, NY, USA, pp. 2123–2138. doi:10.1145/3243734.3243804.

Martin, M., Livshits, B., Lam, M.S., 2005. Finding application errors and security flaws using PQL: a Program Query Language. ACM SIGPLAN Notices 40 (10), 365. doi:10.1145/1094811.1094840.

Maruyama, K., Omori, T., 2011. A Security-Aware Refactoring Tool for Java Programs. Proceedings - International Conference on Software Engineering 22–28. doi:10.1145/1984732.1984737.

Mens, T., Tourwé, T., 2004. A survey of software refactoring. IEEE Transactions on software engineering 30 (2), 126–139. doi:10.1109/tse.2004.1265817.

Opdyke, W.F., 1992. Refactoring object-oriented frameworks. University of Illinois at Urbana-Champaign.

Pewny, J., Holz, T., 2016. Evilcoder: Automated bug injection. In: Proceedings of the 32nd Annual Conference on Computer Security Applications, pp. 214–225.

PHP Documentation, 2021. https://www.php.net/manual/.

PHP repository - backdoor commit, 2021. https://github.com/php/php-src/commit/c730aa26bd52829a49f2ad284b181b7e82a68d7d.

Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H., 2017. Vuzzer: Application-aware evolutionary fuzzing. In: NDSS, Vol. 17, pp. 1–14. doi:10.14722/ndss.2017.23404.

Schreuders, Z.C., Shaw, T., Shan-A-Khuda, M., Ravichandran, G., Keighley, J., Ordean, M., 2017. Security scenario generator (SecGen): A framework for generating randomly vulnerable rich-scenario VMs for learning computer security and hosting CTF events. 2017 USENIX Workshop on Advances in Security Education (ASE 17). USENIX Association, Vancouver, BC.

Schuckert, F., Hildner, M., Katt, B., Langweg, H., 2018. Source Code Patterns of Buffer Overflow Vulnerabilities in Firefox. Proceedings of Sicherheit 2018 107–118. doi:10.18420/sicherheit2018_08.

Schuckert, F., Katt, B., Langweg, H., 2017. Source Code Patterns of SQL Injection Vulnerabilities. International Conference on Availability, Reliability and Security doi:10.1145/3098954.3103173.

Schuckert, F., Katt, B., Langweg, H., 2019. Difficult XSS code patterns for static code analysis tools. In: Computer Security - ESORICS 2019 International Workshops, IOSec, MSTEC, and FINSEC, Luxembourg City, Luxembourg, September 26-27, 2019, Revised Selected Papers. Springer, pp. 123–139. doi:10.1007/978-3-030-42051-2_9.

Schuckert, F., Katt, B., Langweg, H., 2020. Difficult SQLi Code Patterns for Static Code Analysis Tools. Norsk IKT-konferanse for forskning og utdanning – NISK Norsk informasjonssikkerhetskonferanse 2020 (3). https://ojs.bibsys.no/index.php/NIK/article/view/892

Stivalet, B., Fong, E., 2016. Large Scale Generation of Complex and Faulty PHP Test Cases. Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016 409–415. doi:10.1109/ICST.2016.43.

Thomas, S., Williams, L., Xie, T., 2009. On automated prepared statement generation to remove SQL injection vulnerabilities. Information and Software Technology 51 (3), 589–598. doi:10.1016/j.infsof.2008.08.002.

Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs. Proceedings - IEEE Symposium on Security and Privacy 590–604. doi:10.1109/SP.2014.44.

Yamin, M.M., Katt, B., 2022. Modeling and executing cyber security exercise scenarios in cyber ranges. Computers and Security 116, 102635. doi:10.1016/j.cose.2022.102635.

Yamin, M.M., Katt, B., 2022. Use of cyber attack and defense agents in cyber ranges: A case study. Computers & Security 122, 102892. doi:10.1016/j.cose.2022.102892.

Felix Schuckert is a PhD candidate at the Norwegian University of Science and Technology in cooperation with the University of Applied Sciences in Constance. He received the bachelor's degree in software engineering and a master's degree in modeling and software software engineering at the University of Applied Science in Constance. His current topic is about opportunities of Insecurity Refactoring for training and software development. He is interested in static code analysis, web security, software development and source code patterns of vulnerabilities.