Softwares may reuse some of the components from the source code, therefore, if a vulnerability exists in those components it may also exist in the newer version, since these vulnerabilities are not easy to fix, we would face a problem which is called “Vulnerability Prevalence”.

This paper presents a solution to automatically detect a vulnerability in a given source code.

The system is called Vulnerability Pecker (VulPecker).

Two approaches are used for vulnerability detection:

1. Vulnerability patterns
2. Code similarity

The first one relies on multiple instances of a vulnerability to come up with a pattern.

The second one needs only one, and because it is often thought that programs that behave similarly tend to have similar or the same vulnerability (if there are vulnerabilities).

Code Similarity Algorithms can be at Code-Fragment Level, Code Presentation and Comparison Method.

At Code-Fragment Level there are five ways to define the unit at which we compare programs:

1. Patch-without context (a fragment is all the continuous lines that start with “-”)
2. Slice (it is based on Program Dependence Graph)
3. Patch-with context (a fragment is all the continuous lines starting with “-” and lines with no prefix)
4. Function (a function is a unit itself)
5. File/Component (a file/component is a unit itself)

At Code Representation each fragment can be represented by:

1. Text (not useful because represents few semantic information)
2. Metric (a vector of features is used to compare fragments)
3. Token (a lexical analyzer transforms the source into tokens, with tokens being a line or a component in a line)
4. Tree (a tree shows the syntactic structure of the tokens in source code, such as variables, function calls, etc.)
5. Graph (similar to Control Flow Graph, a function expression is a node, and an edge shows a control flow, control dependency or data dependency)

At Comparison Method, the comparison is either vector based or approximate/exact matching:

1. Vector-based: compare to vectors of the representation of a vulnerability and the representation of a source code.
2. Approximate/exact matching: uses substring matching subgraph matching (isomorphic subgraph)

Code Representation and Code-Fragment levels can be of use when generating the signature of a vulnerability.

VulPecker defines features for vulnerabilities and code reuses:

1. For Vulnerability: Uses diff which consists of multiple diff hunks and for diff hunks we have basic features and patch features. Patch features are of five types:
    1. Whitespace, format or comment
    2. Changes of variables, constants, and functions in a component
    3. If condition, for condition and assignments in an expression
    4. Addition, deletion and movement in a statement
    5. Changes in functions
2. For Code Reuse:
    1. Exact cloning
    2. Rename cloning
    3. Near miss cloning
    4. Semantic clones

VulPecker consists of two phases:

1. Learning Phase: gets three databases as its input, NVD, VPD, VCID.

   These three are used in selection of Code-Similarity algorithm and generating the vulnerability signatures, which builds the outputs (Common weakness enumeration to algorithm mapping and Vulnerability signatures) for the second phase.

2. Detection Phase: works on the output of the first phase and target programs.

   Based on the algorithm mapping, it generates signatures from the target program and the vulnerability engine gets vulnerability signatures as well as the target program signatures and using the algorithm it detects the vulnerabilities and outputs a file containing locations of the vulnerabilities.

A Code Similarity Selection Engine selects an algorithm ("Good Algorithm") that can recognize whether the code is patched or unpatched. After the vulnerability diff hunks are classified then it selects a good algorithm, and for the second applies algorithms based on their code-fragment level and among those types of code-fragment levels slice can be more useful and VulPecker uses the lines that were deleted as the slice where the vulnerability lies. And at the third step the one with the lowest fault rate (false-negative) is selected.

VulPecker also has some limitations such as it only operates on open source products and operates only on the source code, it currently can operate on C/C++ programs.