VFQL: Combinational Static Analysis as Query Language[*]

Guang Chen
School of Software,
Tsinghua University
Beijing, China
chenguan14@mails.tsinghua.edu.cn

Yuexing Wang
School of Software,
Tsinghua University
Beijing, China

Min Zhou
School of Software,
Tsinghua University
Beijing, China

Jiaguang Sun
School of Software,
Tsinghua University
KLISS, BNRist
Beijing, China

ABSTRACT

Value flow are widely used in static analysis to detect bugs. Existing techniques usually employ a pointer analysis and generate source sink summaries defined by problem domain, then a solver is invoked to determine whether the path is feasible. However, most of the tools does not provide an easy way for users to find user defined bugs within the same architecture of finding pre-defined bugs. This paper presents VFQL, an expressive query language on value flow graph and the framework to execute the query to find user defined defects. Moreover, VFQL provides a nice GUI to demonstrate the value flow graph and a modeling language to define system libraries or user libraries without code, which further enhances its usability. The experimental results on open benchmarks show that VFQL achieve a competitive performance against other state of art tools. The result of case study conducted on open source program shows that the flexible query and modeling language provide a great support in finding user specified defects.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**;

KEYWORDS

Static analysis, value flow graph, domain specific language

ACM Reference Format:
Guang Chen, Yuexing Wang, Min Zhou, and Jiaguang Sun. 2019. VFQL: Combinational Static Analysis as Query Language. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3293882.3338997

1 INTRODUCTION

Static analysis is widely used in improving software quality, reliability and security. Large companies use certain static analysis tools as de-facto standard of certifying that the code quality is good enough to be checked-in into version control. There has been extensive studies in the direction of static analysis [1, 2, 4–6].

Static analysis tool is built upon lots of underlying algorithms, logical theories and optimizations. To detect bugs in real world software, the algorithms should be carefully designed and tuned. In modern software development, defect patterns evolve fast and static analysis tools need adapt to the evolution simultaneously. For instance, in static analysis of smart contract (program that runs in Ethereum), users may define *re-entrance* as a defect pattern for interest [7]. To support defecting re-entrance defects, we hope that underlying analyses (including pointer analysis, path reachability etc.) were not to be re-designed. It is preferred that existing static analyses can be re-configured and combined at low cost when supporting new defect patterns. Lots of defect patterns can be formulated as a featured path or node in the control flow or value flow graph. The defect of "returning stack address" is a value flow path which starts from a stack address and ends at returning it in the same function. It is a typical defect (CWE-562) that may cause accessing of invalid memory and crashing the program. Detection of such defect requires interprocedural context sensitive pointer analysis and path sensitive reachability analysis. We will show that it (as well as many other defect patterns) can be defined and checked at very low cost in our framework.

In this paper, we propose an extensible tool for re-using and integrating exists algorithms to find defects of existing or new patterns in C project. A group of domain specific language (called VFQL) is provided for configuring and combining these analyses at low cost. It supports new defect patterns by writing few lines of query in VFQL. The experiment results show that our tool reaches a high precision in Juliet Test Suite and it is extensible to detect real world bugs in open source projects

2 VFQL

2.1 System Overview

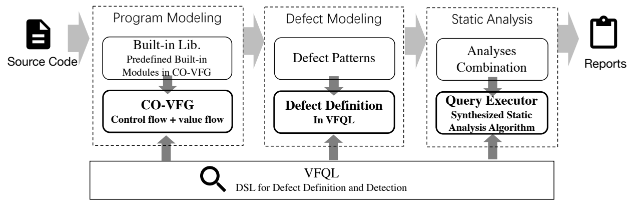

Figure 1: System Overview of VFQL

VFQL takes a C project as input with optional user provided module definitions. Then it captures its build sequence to preprocessing the source files to merged LLVM IR files for each build

[*]This research is sponsored in part by the NSFC Program (No. 61527812), the National Science and Technology Major Project of China (No. 2016ZX01038101), the MIIT IT Funds of China (Research and Application of TCN Key Technologies), the National Science and Technology Support Program of China (No. 2015BAG14B01-02), and the National Key R&D Program of China (No. 2016QY07X1402).

ISSTA '19, July 15–19, 2019, Beijing, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3338997

target. Figure 1 gives the working flow of VFQL, which contains 3 phases: (1) **Program Semantic Modeling**: The semantic model as a graph is extracted from IR file based on a light-weight interprocedural pointer analysis. It will label control and data dependencies in edges. Control flow orders are also augmented in the graph as links. (2) **Defect Modeling**: It accepts user written VFQL query and is synthesized as an algorithm to executed. (3) **Query Execution**: The static analysis algorithm synthesized from the VFQL query is executed on the graph. It combines existing analyses to detect the corresponding defect efficiently.

2.2 Program Semantic Modeling

In VFQL program is represented as a combination form of control flow graph and value flow graph. The former information is used to qualify the execution order of certain actions while the latter information is used for sparse analysis of value path related defects. They are synthesized to a the form of control ordered value flow graph (CO-VFG for short). CO-VFG contains a number of modules(functions) in program. Each module of CO-VFG is an intra-procedural value flow graph with control flow information, i.e., nodes are value flow entities labeled with properties, edges determine the value flows and its conditions, the control flow information are added as link between nodes based on their def-site location's predecessors and successors. Primitive labels of CO-VFG are defined by node type: (1) **const**: represent an LLVM const, (2) **copy**: a value that is copied from another one, (3) **operator**: a value that is the result of computation over other values, and (4) **join**: merge node of more than one values.

Compared with similar approaches such as [5, 6], CO-VFG has the following characteristics: (1) It is augmented with control dependencies (annotated on nodes) and data dependencies (annotated on edges), dependency conditions are further defined based on nodes. (2) Moreover, control flow order is also augmented. Each node is also linked from its control flow predecessors and to successors, in order to query control flow order related patterns.

Construction of CO-VFG is based on the result of pointer analysis. Flow and field sensitive, path insensitive interprocedural pointer analysis based on access path is employed to compute point-to summary for each function [2]. The construction procedure is based on the CPA framework [1]. Similar to construction of TGSA [3], data dependency is annotated as conditions on edges that entering a join node (e.g., γ node in Figure 3). Then two important issues are addressed : (1) weak update, (2) function calls.

```
void foo(int* x, int* y, unsigned char p) {
        int *a;
        a = (p) ? x : y;
        (*a) = 1;
}
```

Figure 2: Example of Weak Update

A pointer may have multiple point-to targets and weak update is necessary when the content in target memory is updated via the pointer. In that case, the updated value as well as its decisive condition should both be represented in the graph. For instance, a tricky tiny example is given in Figure 2. The content of either x or y is updated via the pointer a depending on the condition p. Its CO-VFG is shown in Figure 3. In the graph, x is updated to 1 under the condition $N31 = N4$. This type of condition is called formal condition. And the equivalent condition $N24$ in this example is its final condition. The trick here is that no matter how complex the final condition will be, the formal condition is not changed. We leave the computation of final conditions in analysis phase , which saves the cost of CO-VFG construction phase. As for function call, we treat every memory location that has been read or modified as formal inputs and outputs. Interprocedural calls are organized by call site structure in the CO-VFG where actual and formal inputs / outputs are connected. Moreover, as we have already done pointer analysis, function call via function pointer is also explicitly modeled.

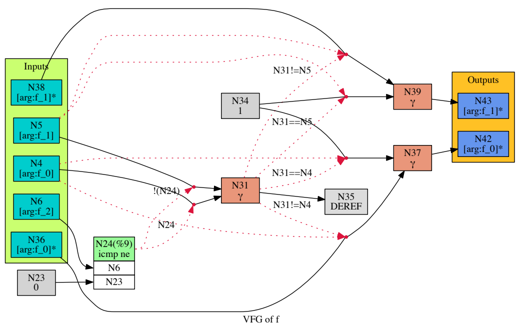

Figure 3: CO-VFG of Weak Update

```
// for malloc function
module malloc(s)
  inputs: s
  outputs: $ret, $ret*
  flow:
    null -> $ret
    heap -> $ret
```

```
// for free function
module free(p)
  inputs: p, p*
  outputs: p*
```

Figure 4: Modeling of Built-in Functions in VFQL

Finally, control dependency is annotated on nodes, then nodes are threaded by control flow order. Loops are unwound explicitly, which greatly simplifies the analysis but causes majority of unsoundness. For library functions without implementation, the semantic of these functions is given as an abstract model which concerns the inputs and outputs. Most common library functions (e.g., `malloc`, `free`) are modeled and provided as a built-in module definitions as in Figure 4. Users can define (or override, for instance, one can assume `malloc` never returns null) for his own modules.

2.3 Defect Modeling

Definition of defects in VFQL consists of two layers: (1) Label layer is used for assigning semantic tags to nodes in CO-VFG. For instance, **const**, **operator** are primitive tags that are already defined (as shown in Figure 5). Other tags can be computed from existing ones with a properly defined label function. (2) In defect pattern layer, defects are defined as value flow paths with constraints. It is the major part of VFQL.

trace	:=	label quantifier label [**where** label] [**in** *id*]
label	:=	**label**(*id*) |
		not label | label **and** label | label **or** label
quantifier	:=	**to** | **all to**

There are two kinds of queries. The result of *trace* with quantifier "`to`" is a set of paths, the start and end node of each path satisfy given label constraints. The quantifier "`all to`" yields a set of trees, rooted at a desired node and each leaf node of the tree satisfies required label constraints. The clause "`where`" specifies the label constraints to meet along the path. The clause "`in`", if given, limits the query so that it only applies in some function. Then we define a query as "`query [graph option] [solver option] (not) formula`". The result of query is a subset of *trace* where after each item of *trace* is converted to an SMT expression, only those (or its negation) whose reachability result meets *solver option* is kept. The *solver option* can be `sat` or `unsat`, which means the expected reachability result is satisfiable or unsatisfiable. Examples of query for common defects are given in Figure 6. The *graph option* (vf or cf, default to vf) determines whether the edges of path are value flow edges or control flow links in CO-VFG.

```
// label definition for llvm value
null            :      -> label(null);
load ptr        : ptr  -> label(deref);
store val, ptr  : ptr  -> label(deref);
alloca _        :      -> label(stack);
ret val         : val  -> label(return);
...
// label definition for built-in function calls
call free(ptr):
  input(ptr*)  -> label(to_free),
  output(ptr*) -> label(freed);
call malloc: -> label(heap) or label(null);
```

Figure 5: Example of Label Functions

```
// null pointer
query sat label(null) to label(deref)
where not label(operator);
// double free
query sat label(freed) to label(to_free)
where not label(operator);
// use after free
query sat label(freed) to label(copy)
where not label(operator);
// return stack address
query sat label(stack) to label(return);
```

Figure 6: Example of Query Common Defects

2.4 Query Execution

Query is executed in two phases: path collecting and constraint solving. In the first phase, all possible paths are explored using a memorized depth first search. Summary is computed to avoid repeated search. Paths that satisfy the query criterion are collected as potential defect paths. Then the path condition, as well as all its control/data dependency condition, is formulated as a logical formula and is checked for satisfiability using various methods (including pattern matching, linear solving or SMT). To reduce time consumption in reachability checking, one can use linear constraint solving or conflict nodes matching, which may produce imprecise result but run much faster. If precision is of concern, user can specify SMT solver as the preferred way of reachability checking.

2.5 Usage

VFQL can be invoked with the following command:

```
vfql -source SPATH -module MPATH -load-module MODULES
- SPATH  : path to source code
- MPATH  : path to module definitions
- MODULES: names of the module to load
```

VFQL accepts three command line arguments. The first argument *-source* specifies the location of source code. *-module* argument is a list of module definition files to read. The last argument *-load-module* is optional. By default, all modules in the module files and built-inf modules are loaded. When *-load-module* argument is given, only the given modules are loaded. VFQL first captures the build sequence and preprocesses sources files to merge IR files. Then it begins constructing CO-VFG. When finished, the main interface of VFQL appears, as shown in Figure 7. There are mainly three panels in the GUI. The left panel is CO-VFG Viewer Panel, the upper panel is VFQL Editor Panel and the right panel is Defect Viewer Panel. The CO-VFG Viewer Panel will list the modules in the project and display the graph of selected module. The VFQL Editor panel allows the user to enter the query. It has an *Execute* button which will execute the query when clicking. Then the summary will be computed and feasible paths result will be listed in the Defect Viewer Panel. Clicking a defect id in the list will show the defect value flow trace. The button *Show Defects* in the VFQL Editor Panel will display the source code path in web page.

3 EVALUATION

The evaluation of our tool is done on three benchmarks and two open source projects. The experiment is conducted on a ubuntu 16.04 with 32GB RAM.

3.1 Results on Benchmark

To evaluate correctness of our tool on Juliet Test Suite[1], we first define the query corresponding to the selected defect types (null pointer dereference, double free, use after free) as shown in 6. We select three open source state of art static analysis tools for comparasion: clang static analyzer(CSA), CPPCheck, and infer. #**TP** measures the recall rate of a tool and #**TP**/#**Rep** measures the precision. The result is shown in 1. The result shows that we reach a higher recall rate and precision over all tools, though we are not aiming at comparing them. The reason of the result is that the data dependency of each value flow as well as the update condition of pointer is modeled accurately in the graph (which is lazily computed to avoid overhead). And we use SMT solver to accurately solve the constraints. Other tools either use pattern matching or does not invoke SMT solver.

3.2 Case Study

The first case study comes from a chat client IRSSI which has an CVE entry.[2] The cause of the bug is that *localtime* may produce

[1] https://samate.nist.gov/SARD/testsuites/juliet/
[2] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-10965

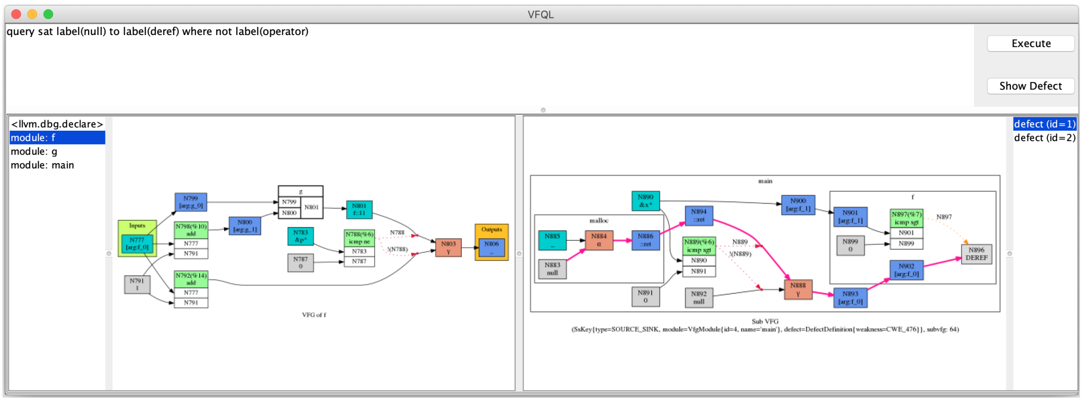
```
query sat label(null) to label(deref) where not label(operator)
```

Figure 7: GUI of VFQL

Table 1: Results on the Juliet Test Suite. #Rep is the number of reports and #TP is the number of correct reports.

Type	Total	VFQL		CSA		CPPCheck		infer	
		#Rep	#TP	#Rep	#TP	#Rep	#TP	#Rep	#TP
Double Free	228	228	228	59	59	133	133	62	62
Use After Free	138	138	138	6	6	0	0	23	23
Null Pointer Dereference	270	252	252	54	53	97	37	68	66

nullable value and *asctime* will return null if the argument is null. In *g_strdup*, the argument is dereferenced without any check, thus causing a null pointer dereference. The purpose of this case is to show how to use the model language to find defects when calling library functions. In the module definition in figure 8, the input of function *g_strdup* is labeled with *deref*, the input of *asctime* is connected to its output return value, and the output of *localtime* has a source value labeled with *null*. Then we execute the null pointer dereference query, which successfully find this bug trace.

The second case is from OpenSSl. In OpenSSL, there are functions

```
module asctime(arg)                module localtime(arg)
  inputs: arg                        inputs: arg
  outputs: $ret                      outputs: $ret
  flow:                              flow:
    null -> $ret when(arg==null)       null -> $ret
    _    -> $ret when(arg!=null)       _    -> $ret

input(g_gstrdup)[0]                module g_strdup(arg)
  -> label(deref)                    inputs: arg
                                     ouputs: $ret
```

Figure 8: Module Definition For IRSSI

whose return value has the meaning whether the call succeed or not. However, we notice that not all there return values are checked. In this case study, we show how to use the query language to find user defined defects such as missing checks on some specified values. The label function and the query is shown in figure 9. One of the querying results is shown in figure 10.

4 CONCLUSION AND FUTURE WORK

This paper describes the flexible bug detection frame work VFQL with customizable modeling and query language support.VFQLshows

```
// openssl.label
call RAND_bytes: -> label(ret_RAND_bytes);
// built-in labels for icmp instruction
icmp l, r: l -> label(check), r -> label(check);
// the query
query unsat not (label(ret_RAND_bytes) all to label(check));
```

Figure 9: Label Function and Query Used for OpenSSL

```
for (count = 0; COND(c[D_RAND][testnum]); count++)
RAND_bytes(buf, lengths[testnum]);
return count;
```

Figure 10: Query Result Example

high correctness rate compared to existing tools. The usefulness is shown in two case studies in two real world projects. The underlying representation and our language is extensible.

REFERENCES

[1] Beyer, D., and Keremoglu, M. E. Cpachecker: A tool for configurable software verification. In *International Conference on CAV* (2011), Springer, pp. 184–190.
[2] Cheng, B.-C., and Hwu, W.-M. W. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. *Acm Sigplan Notices 35*, 5 (2000), 57–69.
[3] Havlak, P. Construction of thinned gated single-assignment form. In *Languages and Compilers for Parallel Computing, 6th International Workshop, Portland, Oregon, USA, August 12-14, 1993, Proceedings* (1993), pp. 477–499.
[4] Machiry, A., Spensky, C., Corina, J., Stephens, N., Kruegel, C., and Vigna, G. {DR}.{CHECKER}: A soundy analysis for linux kernel drivers. In *26th {USENIX} Security Symposium ({USENIX} Security 17)* (2017), pp. 1007–1024.
[5] Shi, Q., Xiao, X., Wu, R., Zhou, J., Fan, G., and Zhang, C. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on PLDI* (2018), ACM, pp. 693–706.
[6] Sui, Y., and Xue, J. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction* (2016), ACM, pp. 265–266.
[7] Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., and Vechev, M. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM, pp. 67–82.