MACKE: Compositional Analysis of Low-Level Vulnerabilities with Symbolic Execution

Saahil Ognawala[1], Martín Ochoa[2], Alexander Pretschner[1], Tobias Limmer[3]
[1] Technical University of Munich, Germany, {ognawala,pretschn}@in.tum.de
[2] Singapore University of Technology and Design, Singapore, martin_ochoa@sutd.edu.sg
[3] Siemens AG, Germany, tobias.limmer@siemens.com

ABSTRACT

Concolic (**conc**rete+symb**olic**) execution has recently gained popularity as an effective means to uncover non-trivial vulnerabilities in software, such as subtle buffer overflows. However, symbolic execution tools that are designed to optimize statement coverage often fail to cover potentially vulnerable code because of complex system interactions and scalability issues of constraint solvers. In this paper, we present a tool (MACKE) that is based on the modular interactions inferred by static code analysis, which is combined with symbolic execution and directed inter-procedural path exploration. This provides an advantage in terms of statement coverage and ability to uncover more vulnerabilities. Our tool includes a novel feature in the form of interactive vulnerability report generation that helps developers prioritize bug fixing based on severity scores. A demo of our tool is available at https://youtu.be/icC3jc3mHEU.

CCS Concepts

•Security and privacy → Vulnerability management; •Software and its engineering → Software testing and debugging; •General and reference → *Verification;*

Keywords

Symbolic execution, Compositional analysis

1. INTRODUCTION

Symbolic execution has been used for analyzing programs and to look for vulnerabilities of the kind that are typically hard to find for "blackbox" methods that ignore specific program structure. Symbolic execution performs much better in terms of coverage [27], finding bugs in parts of the code that are seldom exposed via random testing. This can be attributed to the fact that symbolic execution exploits the semantics of the program by assuming symbolic values for the input parameters and simulating possible execution paths. But symbolic execution suffers from bottlenecks of underlying model checkers and constraint solvers [14, 29]. Since most of the real-world programs are highly intricate and contain many environmental interactions, the size of the constraints (path conditions) generated during symbolic execution may grow too large for constraint systems to solve in a reasonable amount of time. This leads to low coverage of the program, potentially leaving many vulnerabilities undetected.

In this paper, we present a tool that enables testers to detect *low-level vulnerabilities* (defined, for this study, as unhandled memory operations resulting in memory out-of-bounds/buffer overflow) in a program using symbolic execution in a reasonable amount of time. "Reasonable" amount may be defined in terms of time taken for program analysis, or required computing resource. However, for this study, we will perform our performance comparison in terms of time taken for the full analysis, only. We achieve our goal by performing a fully compositional[1] analysis of the program under test. Our tool, named Modular And Compositional analysis with KLEE Engine (MACKE[2]), makes use of symbolic execution techniques at the level of C functions, and then combines the results using static code information and inter-procedural path feasibility. Moreover, our tool allows security experts to reach informed decisions on fixing vulnerabilities based on their respective severity scores and potential risk.

Problem: Most symbolic execution tools generate test cases by starting at the entry point of the program (forward symbolic execution), resulting in insufficient code coverage. This leaves many potential bugs undetected. On the other hand, symbolically executing only individual functions, f, yields many "false positive" vulnerabilities, which may never materialize if the corresponding inputs are sanitized by the functions that (transitively) call f. Compositional approaches to symbolic execution, such as [9, 10, 16], have either not been evaluated on multiple real-world programs or are not accompanied by automated tools.

Solution: Our solution is a three-step approach – Firstly, MACKE performs symbolic execution on the individual components of a program, in isolation. This has the advantage of higher code coverage and ability to uncover many low-level vulnerabilities in all program components. Secondly, MACKE uses results of the first step to reason about (and, therefore, reduce the number of) reported vulnerabilities from a compositional perspective, i.e. by finding feasible inter-procedural paths for those vulnerabilities to be exploited. Thirdly, MACKE assigns severity scores to reported vulnerabilities by considering several characteristic features and provides the result in an interactive visual format.

Contribution: In terms of compositional analysis of vulnerabilities, the contribution of our work is three-fold – (i) evaluation on multiple real-world examples, which is missing in

[1]We will use "components" and "functions" interchangeably since MACKE works on C code only.

[2]Tool available at https://github.com/tum-i22/macke.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

ASE'16, September 3–7, 2016, Singapore, Singapore
ACM. 978-1-4503-3845-5/16/09...
http://dx.doi.org/10.1145/2970276.2970281

```
3  int mask_b(int* b, int n) {
4      b[n++] = 1; /* potential buf. overflow */
5      return n;
6  }
7  int main(int argc, char** argv) {
8      int i, n=0, b[4] = {0, 0, 0, 0};
9      for (i=0; i<argc, i++) {
10         if (*argv[i]=='b')
11             n = mask_b(b, n)
12         else
13             foo(); /* expensive function */
14     }
15     while(1) {
16         if (getchar()) /* symbolic input */
17             /* ...do something... */
18     }
19     return 0;
20 }
```

Listing 1: Program to show effectiveness of targeted-search.

[1, 10], (ii) automation of all stages of instrumentation and concolic execution, which is missing in [10], and (iii) an open–source implementation for reproduction of results, which is missing in [9, 16, 10, 19]. We also provide a compositional approach to ranking vulnerabilities on perceived severity scores, which is absent from all the previously cited works.

This paper is structured as follows. In Section 2 we describe our approach, starting with KLEE, the underlying symbolic execution engine, and the architecture of MACKE. We provide preliminary evaluation results in Section 3. Section 4 enumerates some related work to our tool. In Section 5 we conclude our paper.

2. APPROACH

As discussed in the previous sections, termination of forward search strategies in symbolic execution depends on the ability of the underlying constraint solver to return quickly and effectively [14, 29]. In our solution, we prefer to reduce the size of constraints that we input to the constraint solvers. Then we use compositional methods to combine the results. Before explaining MACKE, we describe KLEE, the symbolic execution engine used by our tool.

2.1 KLEE - A Symbolic Execution Tool

KLEE [4] is the most popular symbolic execution tool and it is well documented and maintained by its developers. This is the reason we chose it as the symbolic execution engine for MACKE framework. KLEE is a coverage-first symbolic execution tool, which means that it focuses on covering as many paths in a program as possible. This is done by symbolically executing a program till a branching statement is encountered. At this point, the branching condition (p) is analyzed to determine its feasibility, depending on the path condition (PC) obtained from all the preceding branching conditions in the path and assignments to variables in the branching conditions. A PC is defined as a conjunction of predicates that represent which branch (true or false) was taken at any branching statement. If both branches (p and $\neg p$) are feasible, then the program is cloned with both possibilities added to the PC, respectively for each clone [4].

A path is said to *end* at a node when (i) the next branching condition to be added is inconsistent with the PC, (ii) an exception is with the current PC and a possibly unsafe memory handling operation, or (iii) the path has reached a leaf node with a return statement. At the end of the path, an attempt is made by the underlying decision procedure to find an assignment for the symbolic variables in the PC. By default, this is done by a satisfiability solver such as STP or any other constraint solver [15, 13].

2.2 MACKE

MACKE is a framework written on top of the KLEE symbolic execution engine for compositional analysis of C programs. The complete procedure of compositional analysis is divided into three stages, each of which we describe next.

2.2.1 Looking for Low-level Vulnerabilities

The first step of such a compositional analysis is the isolation of *low-level components*. Low-level components may be defined differently for different programming languages or runtime frameworks. For all experiments in our study, these low-level components are C functions.

To look for buffer overflows in low-level components, MACKE's *static analyzer* isolates them and creates a *unit-test file* for each of them. These isolated components are then symbolically executed by *KLEE* to obtain test cases and buffer overflow violation reports for each C function. A benefit of symbolically executing isolated components is that this process may be parallelized efficiently. As our intent is to focus on inter-procedural interactions [1] only in the second step (Section 2.2.2), this approach makes sense in the first step. When functions are isolated, the function calls are *not* stubbed with symbolic return values but are executed normally. Doing this, in our experience, results in many false positives to a degree that does not provide a good cost-benefit w.r.t. higher path coverage in the isolated component. Also, doing this obviates application of *static compositional analysis* step, as described in Section 2.2.2.

Referring to the code in Listing 1, which we will use as a running example[3] [4], this means that we first isolate functions `main`[5] and `mask_b` and then execute them both with symbolic arguments (`argc` and `argv` for `main` and `b` and `n` for `mask_b`). *Symbolic arguments* are the variables which determine the execution paths in symbolic execution. As the output of this stage, we get unit test-cases for both functions individually. It is highly likely that we achieve full path coverage in `mask_b` due to only two non-expensive instructions.

Some covered paths lead to memory out-of-bounds error (buffer overflow), based on some assignment to the symbolic arguments. These test-cases are reported (in *unrefined bug reports*) as low-level vulnerabilities, or simply *bugs*. In Listing 1 such a vulnerability exists on line 2. Function `mask_b` might try to write outside the bounds of array `b`. The same vulnerability would be reported in `main` function if more than 4 elements of `argv[0]` are 'b', and line 9 is executed.

2.2.2 Exploring Paths to Vulnerabilities

After we have a report of bugs found by symbolic execution on the isolated functions, the next step is to rule out the ones that are *unfeasible*, i.e. they cannot be reached due to input sanitization conditions in higher level functions.

Below are the activities that MACKE perform for exploring paths to low-level vulnerabilities – Firstly, *static analyzer*

[3]Program directly adapted from *shortest distance symbolic execution* (SDSE) description in [26].

[4]Include statements are not shown, so lines start from 3.

[5]`main` is treated the same as all other functions. MACKE does this by changing `main`'s function name to `main_aux`.

```
Error: memory error: out of bound pointer
File: /home/klee/direct_path.c
Line: 4
assembly.ll line: 214
Stack:
        #000000216 in mask_b (b=47144544, n) at /home/klee/direct_path.c:4
        #100000306 in __user_main (argc=1, argv=45140464) at /home/klee/direct_path.c:37
        #200001826 in __uClibc_main (main=26444192, argc=1, argv=45140464, app_init=0, app_fini=0, rtld_fini=0,
stack_end=0) at /home/klee/klee_build/klee-uclibc/libc/misc/internals/__uClibc_main.c:401
        #300003337 in main (=1, =45140464)
```

(a)

```
Error: memory error: out of bound pointer
File: /home/klee/direct_path.c
Line: 4
assembly.ll line: 214
Stack:
        #000000214 in mask_b (b=23407664, n=4) at /home/klee/direct_path.c:4
        #100000257 in main_aux (argc=6, argv=42976656) at /home/klee/direct_path.c:11
        #200000296 in __user_main (argc=5, argv=22519216) at /home/klee/direct_path.c:32
        #300001813 in __uClibc_main (main=21734032, argc=5, argv=22519216, app_init=0, app_fini=0, rtld_fini=0,
stack_end=0) at /home/klee/klee_build/klee-uclibc/libc/misc/internals/__uClibc_main.c:401
        #400003324 in main (=5, =22519216)
```

(b)

Figure 1: KLEE bug reports – (a) describes a bug in `mask_b` and (b) describes a matching bug in `main`.

creates a *call-graph* and *control-flow graph* of the program. Secondly, MACKE analyzes the call-graph in combination with the unrefined bug reports generated in the previous stage. The bug reports that KLEE produces contain relevant details, viz. the problematic symbolic variable(s), the problematic value(s) those variables take (function exploit), the source line containing affected instruction, and the call stack up to function containing the affected instruction.

With these artifacts, the second stage of compositional analysis may be further divided into following sub-steps – Static compositional analysis and partial PC matching.

Static compositional analysis: The first step in exploring vulnerability paths aims to confirm whether some of the bugs reported at the isolated function level can be reproduced via higher compositional level. MACKE does this by comparing the location of a bug reported in a function, f, to that reported in the parent function of f. Consider the call-sequence of the program in Listing 1. For every function, such as `mask_b`, that contains bug(/s) at the isolated level, we look at bug reports of all functions that call `mask_b`, such as `main`. A bug in f is said to be matching to a bug in parent of f if the call stack of the parent function's bug report shows that the affected instruction is located in the same source file and line as that reported for the bug in f. As shown in Fig. 1(b), the bug reported by KLEE in `main` is at the same program location as the bug reported in `mask_b`, as shown in Fig. 1(a). Thus, these two bugs, as reported by KLEE in isolated components, are called matching bugs.

If a matching bug is found to be reported in calling function, we recursively do a similar static bottom-up reasoning all the way up to the entry point of the program (in this case `main` is the entry point of the program). We define "lowest-level" function as the one which does not call any other function, or calls functions external to the tested system.

The above described initial compositional analysis step confirms reachability of some (or all) of the vulnerabilities reported in the isolated functions. However, due to time-outs in constraint solver, this does not help us in *ruling out* reproducibility of the bugs for which matching bugs are not found. This gives rise to the need for partial PC matching.

Partial PC matching: To understand the need for partial PCs, let us reiterate the cases when a matching bug may not be found in a higher compositional level - a) if input to the lower level component is sanitized in a higher level component, or b) if the higher level component is incompletely covered (time-out). In case a), there is nothing to report as the malicious input is already taken care of. In case b), the set of partially covered paths in `main` are called *partial PCs*.

Consider again Listing 1. Assume that a matching bug

```
7  int main(int argc, char** argv) {
8      int i, n=0, b[4] = {0, 0, 0, 0};
9      for (i=0; i<argc, i++) {
10         if (*argv[i]=='b')
11            klee_assert(!(!\\
12            memcmp(*argv,"bbbbb",sizeof(argv))\\
13            && argc==5));
14         else
15               foo(); /* expensive function */
16     }
17     while(1) {
18         if (getchar()) /* symbolic input */
19               /* ...do something... */
20         }
21     return 0;
22 }
```

Listing 2: Modified main function. Call to `mask_b` replaced by assertion statement

to line `mask_b` has not been reported in `main`. Note that `mask_b` has been sufficiently covered to find a vulnerability. One way of reducing the number of paths for symbolic execution to explore in `main` is to replace the call to `mask_b` with the summary of those symbolic execution runs of `mask_b` performed previously, which resulted in bugs. Programmatically, summarizing is done by the *PC Matcher* component of MACKE as follows – i) prepare a KLEE assertion statement that compares actual parameter with solution assignments to formal parameters found by KLEE, ii) replace function call by the KLEE assertion statement.

For the code in Listing 1, MACKE modifies the code, as shown in Listing 2. The values, `"bbbbb"` and `5`, are assignments found for `(b, c)` that lead to the buffer overflow.

Furthermore, the time taken to reach the target compositional interactions can be decreased by executing those branches first that take the execution closest to the target statements. As a part of the full MACKE framework, we implemented an additional search strategy in KLEE, known as *targeted-search*. For our targeted-search mechanism, we draw inspiration from the *best-first strategy* described in [30] and variants of SDSE described in [26, 30]. The PC matching phase of our approach is essentially another run of KLEE on isolated components, but with targeted-search strategy enabled, instead of the default *cover-new-paths-first* strategy. Targeted-search is implemented by, first, picking the shortest path to the function containing the assertion statement (from program call-graph), and, then, employing a source-code based distance metric within the container function. This way, we avoid spending time in expanding those execution paths that do not reach the assertion statements. For the code in Listing 1, symbolic execution will cover line 9 only when the PC is $((i < argc)\&\&(*argv[i] ==' b'))$. Considering that this is true for only a few possible inputs to the program, targeted-search performs better than KLEE's path-search by directing exploration explicitly to line 9.

2.2.3 Ranking the Vulnerabilities

A thorough compositional analysis for finding low-level vulnerabilities is more useful when there is a process to prioritize those vulnerabilities. After consulting with our industry partners, we decided to implement in our framework an interactive procedure to assign severity scores to vulnerable functions that are found in the analysis stages of MACKE. This severity score is based on the functions described below (with their intuition), and a weight (impact factor) between 1 and 5 associated with each function –

Table 1: Results of compositional analysis with MACKE

1	2	3	4	5	6	7	8	9	10
Program	LOC	Coverage		Vuln. Instr.			1-level up	`main` exploit	
		Forward	Compositional	Splint	Forward	Compositional	Compositional	Forward	Compositional
Bzip2	7725	5%	53%	1263	0	106	16	0	0
Grep	10929	44%	54%	3292	0	114	12	0	0
Flex	11784	7%	21%	1137	1	75	9	1	1
Coreutils	63542	43%	51%	10656	20	240	27	20	21

(i) The function $\mathsf{len\_chain}(f)$ returns a natural number representing the depth of function hierarchy through which a vulnerability in f might be exploited. It has the impact factor L. If a function can be exploited through a long hierarchy, it's more likely somebody forgot to sanitize the exploit input. (ii) The function $\mathsf{is\_int}(f)$ returns a boolean according to whether the function f is an exposed interface or not. It has the impact factor I. Vulnerability in an exposed interface, such as the `main` function, is easily exploitable and must be fixed with higher priority. (iii) The function $\mathsf{vuln\_inst}(f)$ returns the number of distinct instructions that were found to contain a vulnerability and has the impact factor N. More vulnerabilities strongly indicates a missing input sanitization check somewhere in the function. (iv) The function $\mathsf{d\_interface}(f)$ returns the proximity (length of nested function chains) of the function to an exposed interface and has the impact factor D. A vulnerable function closer to an exposed interface may be easier to exploit. (v) The function $\mathsf{is\_outlier}(f)$ returns a boolean depending on whether the number of vulnerable instructions found (Section 2.2.1) is much greater than the average number of vulnerable instructions per function in the program[6]. The intuition behind this is the same as that for $\mathsf{vulnerable\_inst}(f)$. It has the impact factor O.

We formulated the above functions with our industry partners and, based on our combined intuitions on the programs that we analyzed, we used the following function, s, to calculate the total severity value:

$$s(f) = L * \mathsf{len\_chain}(f) + I * \mathsf{is\_int}(f) + N * \mathsf{vuln\_inst}(f) + D * \mathsf{d\_interface} + O * \mathsf{is\_outlier}(f)$$

Functions with higher severity scores are, in our view, more vulnerable to attacks. The specific values for the impact factors are also, as the function s itself, dependent on the context of development and vulnerability analysis, as we clarify once more in Section 3.

As a final presentation step, MACKE color codes the ranges of severity scores for all functions in the program and displays the call graph, with function and instruction level details of the test cases that cause a vulnerability to be exposed with compositional analysis.

3. RESULTS

We conducted experiments to show (Table 1) that our compositional analysis technique with symbolic execution performs better than a plain forward symbolic execution technique and naïve static analysis[7], by evaluating the outcomes on a number of parameters. We applied MACKE on *4 open-source applications* and evaluated the results w.r.t. forward symbolic execution over a comparable amount of total time. The programs considered were – Flex, Grep[8] Bzip2 and a set of Coreutils programs (91 Unix utilities). For each candidate program, we put a limit of *2 minutes per function* for the stage one of compositional analysis with MACKE, i.e. looking for low-level vulnerabilities. After this stage, all the static analysis processes and instrumentation for targeted path search were performed by MACKE automatically and took less than *5 minutes per program*. For comparison with forward symbolic execution, we ran KLEE (with `nurs:covnew` as the search method) on the `main` functions for *2 hours per program*.

The source code coverage in all four programs was found to be higher with MACKE compositional analysis (column 4), than forward symbolic execution at `main` functions (column 3). In case of Coreutils and Grep, however, the relatively smaller increase in coverage may be attributed to the fact that most of the functionality in these programs are implemented in single monolithic functions, instead of the more modular implementations found in Bzip2 or Flex. Overall, the increase in coverage can be trivially attributed to the first stage of compositional analysis that looks for low-level vulnerabilities by separately analyzing functions in isolation.

It can be seen from Table 1 that vulnerable instructions reported by MACKE (column 7) are more comprehensive than forward symbolic execution (column 6). However, this number is still far lower than a static code analysis tool (column 5). We infer from these figures that due to higher coverage, compositional analysis finds more pote ntial vulnerabilities (with exploit parameters) in individual components, than forward symbolic execution. However, developers do not have to go through thousands of reported vulnerable instructions, many of which have no corresponding exploit parameters, as is the case with static analysis. In order to further demonstrate the effectiveness of MACKE, column 8 lists the number of vulnerabilities reported in isolated functions, that were also reproducible via *at least one higher level of composition*. This shows that MACKE's compositional approach helped to confirm the reachability of *some* low-level vulnerabilities through higher compositional interfaces, thereby refining the set of reported vulnerabilities even more. Last two columns list the number of vulnerabilities that were reported from the `main` functions. For compositional case (last column) this set is a subset of the vulnerabilities reported in "1-level up". In the case of Coreutils (version 6.10), we found one real vulnerability (exploitable though `main`) in `touch.c`, that could not be found with forward symbolic execution. Finally, in Fig. 2, we present part of an interactive report generated by MACKE for Grep program. All the vulnerable functions are represented in compositional "chains" for depicting their

[6] Specifically, the number of vulnerable instructions $> \mu + 2*\sigma$, where μ is the average number of vulnerable instructions in all functions and σ is the standard deviation in number of vulnerable instructions.

[7] Using Splint for memory management vulnerabilities

[8] Sources for Flex and Grep were obtained from the Software-artifact Infrastructure Repository (SIR)[12].

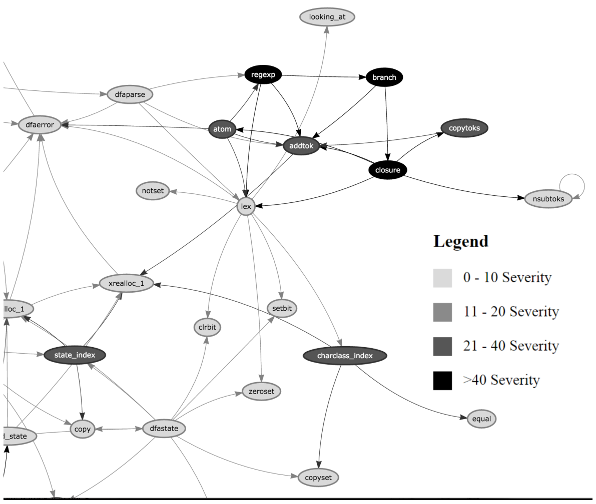

Figure 2: Interactive tool's sample result on Grep (L=3,I=5,N=2,D=4,O=1).

reproducibility – a novel feature in symbolic analysis tools.

We claim that MACKE's approach leads to higher source coverage than forward symbolic execution, more compositional information about reported vulnerabilities than static analysis, a low number of *possible* false positives and highlighted function chains in a graphical report.

4. RELATED WORK

The earliest conceptualization of symbolic execution dates back to 1976 [23]. Over the years, many improvements to the basic symbolic execution techniques and domain-specific implementations have been developed [5, 22, 20]. Tools for symbolic execution have also been developed for several programming languages [2, 6, 4, 31, 7, 11].

The problem of directing path exploration in symbolic execution to specific source locations is addressed in [34, 26, 30, 3, 33]. Unfortunately, most such works do not adequately describe ways to effectively find vulnerabilities. The same shortcoming also applies to papers that deal with the problem of path explosion in symbolic execution. In [24, 32, 8], we find methods of merging, modifying or summarizing program states or individual components. The technique proposed in [28], for instance, partitions the input space such that dynamic execution may execute separate paths of the program that depend exclusively on one input partition. In [18], the concolic execution tool is modified to deal with applications based on a specific grammar for the input, and the constraint solver is changed from a normal buffer based one to a grammar based constraint solver. Above ways are useful in mitigating the path explosion problem so as to increase coverage in a reasonable amount of time with symbolic execution. However, none of these works are accompanied by a tool for discovering and analyzing vulnerabilities, which, we postulate, should follow increased path coverage in the program.

With respect to compositional analysis, to the best of our knowledge, none of the past works describe a freely available tool that finds vulnerable instructions, compositionally analysis them, and assigns severity to vulnerable components, all in a single work. Additionally, previous works like [1, 16, 19, 25] do not report reproducible improvements on existing evaluations [4], such as on Coreutils. Our work shows a clear improvement in terms of program coverage and vulnerability discovery. Our results also indicate a reduction in probable false positives w.r.t. static analysis tools or compositional analysis methods that use static analysis for low-level vulnerability detection, such as [21]. Compared to our approach, pure static code analysis typically reports many more vulnerabilities using only code patterns, most of which, in our experience, can be discarded as false positives without analyzing path feasibility. Some works such as [1, 25] do not limit the directed search strategy to finding vulnerabilities in the code, but to a more generalized goal of generating summaries for parts of program, to be re-used for compositionally analyzing higher-level components.

In [10], the authors describe *verification* of a proprietary Windows library. The proposed technique uses the same first step of executing functions in isolation (without stubbing return values). However, the second step of path exploration is highly tailored to the image parsing library being verified. Specifically, only 12 functions analyzed by the authors were *not* fully covered, compared to more than 510 functions in our analysis. For the impartially covered 12 functions, authors of [10] either manually inlined the functions to the calling contexts, or manually examined the calling pre-conditions to decide absence of memory related faults. For a larger scale evaluation like ours, this would, of course, be infeasible. Moreover, our automation approach for summarizing paths to potential vulnerabilities and automatically replacing calling contexts with assertion statements works for more general scenarios. In a related work by **(author?)** [9] only those low-level functions are (automatically) summarized in the program whose input parameters are free of constraints up to the point that they are called [17]. For the real-world examples that we evaluated MACKE on, this would be unproductive because most of the low-level functions in the call-graphs were dependent on variables that were part of path-constraints up to the calling statements.

With the above research gaps in mind, our work aims to provide an open-source tool to find vulnerabilities, analyze their reachability compositionally, and report vulnerabilities in the context of their usage environment. Even though other works in the past have addressed the problem of vulnerability discovery with a similar compositional approach, they have either, not been shown to be effective (in terms of coverage and discovered vulnerabilities) on multiple real-world programs, or not fully automated or are closed source implementations. Moreover, none of the above works integrate vulnerability discovery with priority based reporting.

5. CONCLUSION

In this paper, we have presented a tool for compositional analysis that uses symbolic execution on the isolated functional level and combines the results using static code analysis and targeted path search. We evaluated MACKE on four open-source projects. In addition to being better than forward symbolic execution in terms of program coverage and vulnerability discovery, MACKE also includes a severity scale that is based on the context around a reported vulnerability, such as the distance of the function from a known interface and the number of possibly vulnerable instructions. Values of impact factors (L, I, N, D and O) are chosen based on the context of development, which is, naturally, specific to the responsible stakeholders. An empirical study of these impact factor values is left to future work. All these impact factors together form a *severity score*. This, we believe, is novel and crucial because, in the absence of this severity metric, it would be very difficult for developers to prioritize the bug fixing procedure. When combined with severity scores, our compositional analysis tool empowers developers to not only analyze the reported vulnerabilities with more contextual information but also reason about which bugs are more critical to be resolved than others. We wish to point here that the results of our study may have been affected due to particularities of the open-source programs we chose that may not be generalizable to a larger class of programs under test. An empirical investigation on how MACKE can effectively impact the productivity of developers and security of the resulting applications is left to future work.

6. REFERENCES

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS*. 2008.

[2] S. Anand, C. S. Păsăreanu, and W. Visser. JPF–SE: A symbolic execution extension to java pathfinder. In *TACAS*. 2007.

[3] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *TACAS*. 2008.

[4] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, 2008.

[5] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Model Checking Software*. 2005.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. *TISSEC*, 2008.

[7] V. Chipounov, V. Kuznetsov, and G. Candea. *S2E: a platform for in-vivo multi-path analysis of software systems*. ACM, 2012.

[8] C. Y. Cho, V. D'Silva, and D. Song. Blitz: Compositional bounded model checking for real-world programs. In *ASE*, 2013.

[9] M. Christakis and P. Godefroid. IC-Cut: A compositional search strategy for dynamic test generation. In *Model Checking Software*. 2015.

[10] M. Christakis and P. Godefroid. Proving memory safety of the ANI Windows image parser using compositional exhaustive testing. In *VMCAI*, 2015.

[11] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *DAC*, 2003.

[12] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *ESE Journal*, 2005.

[13] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, 2004.

[14] I. Erete and A. Orso. Optimizing constraint solving to better support symbolic execution. In *ICSTW*, 2011.

[15] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, 2007.

[16] P. Godefroid. Compositional dynamic test generation. In *ACM Sigplan Notices*, 2007.

[17] P. Godefroid. Micro execution. In *ICSE*, 2014.

[18] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, 2008.

[19] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *ACM Sigplan Notices*, 2005.

[20] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, 2008.

[21] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Usenix*, 2013.

[22] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*. 2003.

[23] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976.

[24] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Acm Sigplan Notices*, 2012.

[25] Y. Lin, T. Miller, and H. Søndergaard. Compositional symbolic execution using fine-grained summaries. In *ASWEC*, 2015.

[26] K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *Static Analysis*. 2011.

[27] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, 2007.

[28] R. Majumdar and R. Xu. Reducing test inputs using information partitions. In *CAV*, 2009.

[29] H. Palikareva and C. Cadar. Multi-solver support in symbolic execution. In *CAV*, 2013.

[30] A. Pretschner. Classical search strategies for test case generation with constraint logic programming. In *FATES*, 2001.

[31] K. Sen, D. Marinov, and G. Agha. *CUTE: a concolic unit testing engine for C*. ACM, 2005.

[32] K. Sen, G. Necula, L. Gong, and W. Choi. MultiSE: Multi-path symbolic execution using value summaries. In *FSE*, 2015.

[33] N. Sinha, N. Singhania, S. Chandra, and M. Sridharan. Alternate and learn: Finding witnesses without looking all over. In *CAV*, 2012.

[34] T. Xie, N. Tillmann, J. De Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN*, 2009.