A Preliminary Study on Open-Source Memory Vulnerability Detectors

Yu Nong
Washington State University, Pullman, USA
yu.nong@wsu.edu

Haipeng Cai
Washington State University, Pullman, USA
haipeng.cai@wsu.edu

***Abstract*—We present preliminary results of a study on memory vulnerability detectors based on (static and/or dynamic) program analysis. Against a public suite of 520 C/C++ programs as benchmarks which cover 14 different vulnerability categories, we measured the performance of five state-of-the-art detectors in terms of effectiveness and efficiency. Our study revealed that with respect to the particular set of benchmarks we chose: (1) the effectiveness of these studied detectors varied widely: 66.7% to 100% precision, 0% to 100% recall, and 0% to 100% F1 per category, indicating most of the techniques worked extremely well on certain kinds of vulnerabilities yet quite poorly on others, (2) these detectors were generally quite efficient: despite a few outliers, the average (per benchmark) time costs were around one second, (3) except for between the most and least accurate detectors, other pairs of detectors did not have statistically significant and large differences in accuracy in our pair-wise statistical testing. We also share insights into the failures and successes of these detectors obtained from our case studies.**

***Index Terms*—Software vulnerability, detection, comparison, code analysis, precision, recall, accuracy, efficiency**

I. Introduction

The pervasive and diverse presence of software vulnerabilities constitutes a major source of cybersecurity threats. Such vulnerabilities are also costly and highly consequential, having caused a loss of over 1.7 trillion dollars just in a single year [1].

As a response, a growing number of approaches are being developed to detect/discover software vulnerabilities, including penetration testing [2], [3], static/dynamic code analysis [4]–[7], and techniques based on data mining and machine learning [8], [9]. However, there have not been sufficient objective measurements and scientific comparisons of these techniques, which are necessary for systematically understanding the strengths and limitations of available solutions hence developing more advanced countermeasures against ever-diversifying and stealthy vulnerabilities.

Prior work on studying and/or comparing vulnerability defense techniques exists. Yet existing such studies typically evaluate the techniques based on how many vulnerabilities can be found without evaluating with respect to the vulnerability ground truth [3], [10]–[12]. As a result, the evaluation was incomplete in that precision and recall were not both measured. A few other studies focus on discussing detection capabilities comparatively from technical perspectives only, without empirical experiments or assessments [13]–[15]. In [16], the authors presented useful metrics for benchmarking vulnerability detectors without actually performing the benchmarking experiments. In addition, several studies address the study/comparison of vulnerability detection approaches particularly for web services [17] or SQL injection vulnerabilities [18]–[20] only. Among these domain-specific studies, [17], [19] used benchmarks with ground truth hence quantified precision and recall, but only for commercial tools as in [21]. Without knowing the technical details behind tools, it is difficult to deeply investigate and explain tool performance and the potential differences. As it stands, **there is a lack of studies evaluating and comparing open-source vulnerability detectors, based on either static or dynamic analysis or both, reporting complete effectiveness measures (precision, recall, accuracy) while not limited to specific application domains** (e.g., web or database applications).

To fill this gap, we are conducting a large-scale study of open-source vulnerability detectors, including those based on static, dynamic, and hybrid code analysis. This paper reports the methodology of and early findings from evaluating five state-of-the-art detectors [22]–[26] that all focus on memory vulnerabilities (i.e., those compromising memory safety) in C/C++ programs. We target this scope because (1) memory vulnerabilities represent a dominant class of vulnerabilities in modern software, and (2) C/C++ has been by far the most vulnerable language [27] yet also the language in which many critical software systems are written. We have evaluated the five chosen tools against 520 C/C++ benchmarks from a public benchmark suite [21] which cover 14 different categories of memory vulnerabilities, as guided by three research questions:

- **RQ1:** *How effective are these detectors in terms of precision, recall, and accuracy?*
- **RQ2:** *How efficient are the detectors in terms of their cost for detecting vulnerabilities?*
- **RQ3:** *How do these detectors compare in terms of their detection accuracy?*

Our study revealed that, for the benchmarks used, incorporating static analysis in vulnerability detection brought substantial accuracy benefits, especially compared to purely dynamic approaches, mainly because of the low recall of the latter. Meanwhile, different detectors showed varying strengths and limitations for different vulnerabilities: the F1 accuracy ranged wildly from 0% to 100% for individual vulnerability categories, and from 40.5% to 87.3% over all the benchmarks. Also in terms of accuracy, statistically significant and large differences were only found between the most and the least accurate detectors. All the detectors were highly efficient, costing less than one second on average per benchmark for

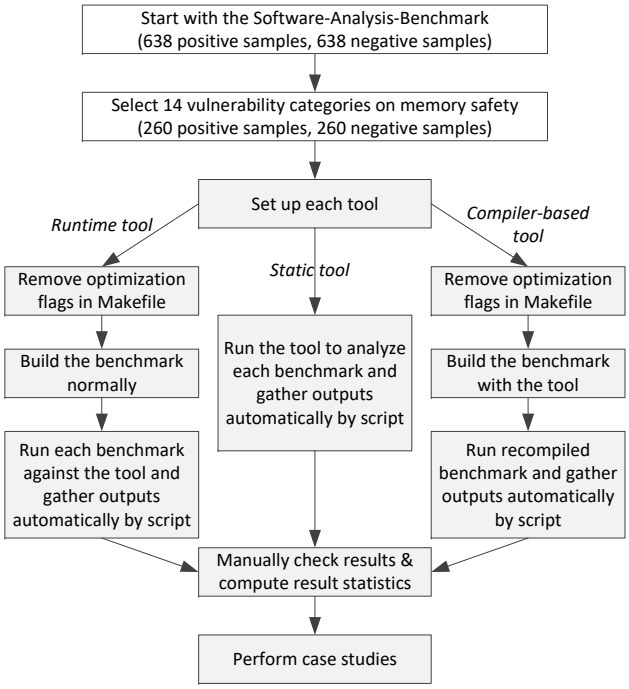

Fig. 1: An overview of our study process flow.

vulnerability detection. We also share insights into varying tool performance obtained from our case studies.

In sum, we contribute preliminary results for understanding where current vulnerability defense techniques are. By focusing on memory vulnerabilities and techniques based on code analysis, our study complements existing peer work in terms of scope and depth. Also, our results also shed light on how techniques of varied nature perform differently and why.

II. Methodology

In this section, we describe the design of our study, including benchmarks and vulnerability detectors used, study procedure followed, and metrics and measures considered.

Overview. Figure 1 gives an overview of the process flow of our study. We started with selecting *benchmarks* that are most relevant to our study focus (i.e., memory vulnerabilities), followed by choosing and setting up vulnerability detection *tools* that are capable of detecting memory vulnerabilities. We ensured the inclusion of both *positive samples* (i.e., benchmarks that are known to be vulnerable) and *negative samples* (i.e., benchmarks that are known to be not vulnerable) for the soundness of the study. Then, we took different approaches for tools of different workings/configurations.

For *static tools* (i.e., those that are based on purely static code analysis and detect vulnerabilities by scanning the code of given programs without recompiling the code), we simply ran each tool against every benchmark (which was automated via scripts we developed) and then manually checked the outputs of the tool to compute statistics of results (i.e., effectiveness and efficiency metrics). For each tool that works via a compiler pass hence requires recompilation of given programs (referred to as *compiler-based tools*), we removed the optimization flags (in order to genuinely test the detector's capabilities), built each benchmark with the tool (which performs the detection analysis during the recompilation), and then ran the rebuilt benchmark so that the tool can produce detection results. This process was also automated through our dedicated scripts. The result examination and metrics computation were done manually the same way as above. The process for *runtime tools* (i.e., those that are based on purely dynamic or hybrid analysis) is similar to that for compiler-based tools, except for that the benchmark needed not to be built with the tools but built normally.

After we finished all the manual results analysis and obtained statistics on tool effectiveness and efficiency, we performed in-depth case studies against chosen cases in order to gain deeper understanding about the effectiveness of the underlying vulnerability detector applied. These understandings allow us to explain why certain tools succeeded or failed in these cases.

Benchmarks. We used the Software-Analysis-Benchmark suite introduced by Shiraishi et al. in [21], which includes 638 positive samples and 638 corresponding negative samples in C/C++—each sample came with a vulnerable version and a corresponding non-vulnerable version (i.e., with the vulnerabilities fixed). While the authors originally curated the benchmarks for evaluating static analysis tools, we intended to see if and how well the vulnerabilities can be detected by tools based on purely dynamic or hybrid analysis as well.

These positive samples covered 51 categories of vulnerabilities (e.g., static/dynamic buffer overrun, stack overflow, data overflow, etc.), out of which we chosen 14 categories that are most relevant to memory safety. Accordingly, we obtained 260 positive samples and the 260 associated negative samples. These 520 C/C++ programs formed the benchmark suite actually used in our study. Each benchmark came with the vulnerability ground truth, which enabled our effectiveness computation.

Tool/detector selection. For our study, we chose five code-analysis-based vulnerability detectors as follows.

- AddressSanitizer [24] is a memory error detector for C/C++ programs. It consists of a compiler instrumentation module based on LLVM [28] and a run-time library that replaces the `malloc` function.
- Valgrind [22] is dynamic binary instrumentation framework. In this study, we used Memcheck, one of the tools built on top of the framework, which detects memory errors.
- MemorySanitizer [25] is a vulnerability detector that can detect uninitialized memory reads in C/C++ programs. It provides a subset of the functionalities of Valgrind/Memcheck but with higher efficiency.
- CBMC [26] is a bounded model checker that detects memory-safety vulnerabilities in C/C++ programs, checking against array bounds, use of pointers, and memory-related undefined behaviors, etc.
- DrMemory [23] is a memory monitoring tool that

identifies memory-related programming errors, such as accesses of uninitialized or unaddressable memory, double frees, memory leaks, and so on.

We chose these tools among others for two reasons. First, we wanted to cover different categories of code analysis approaches underlying the detection techniques, including those based on purely static analysis (CBMC), purely dynamic analysis (VALGRIND, DRMEMORY, MEMORYSANITIZER), and hybrid analysis combining static and dynamic analyses (ADDRESSSANITIZER). In addition, we intended to include at least one state-of-the-art, representative technique in each category. We selected more (three) dynamic approaches because, among the techniques we surveyed, most of those for which we can find publicly available tools were dynamic. Beyond approaches mainly based on code analysis, software vulnerability detection has also been addressed through data mining [8], [9] and machine learning [8] methods.

Metrics and measures. We considered two classes of metrics/measures for our study: *effectiveness* (i.e., performance) and *efficiency* (i.e., cost). For the effectiveness metric, we measured the precision, recall, and F1-measure as the accuracy metric. We computed these effectiveness measures both for each of the 14 vulnerability categories and for all the 520 benchmarks as a whole. For the efficiency metric, we measured the analysis time cost of each detector against each benchmark.

To compute the precision and recall, we examined the output of each detector against each benchmark, so as to manually identify true/false positives/negatives according to the ground truth available for the benchmark. Any vulnerabilities reported in a negative (normal) sample were counted as a false positive, and the sample was counted as a true negative if no vulnerabilities were reported with it. Any vulnerabilities reported in a positive (vulnerable) sample were counted as a true positive, and the sample was counted as a false negative if no vulnerabilities were reported with it. Once we obtained a precision and recall measure, the corresponding F1-measure was computed as 2*precision*recall/(precision+recall).

To compare effectiveness among these chosen detectors, we performed two statistical testing procedures concerning the F1 accuracy of each pair of tools: hypothesis testing to assess the statistical significance of the F1 difference between the pair, and effect size measurement to assess the size of the difference. In particular, we used the paired Wilcoxon signed-rank tests [29] to assess the significance (p value) at the 0.95 confidence level (i.e., $\alpha = .05$), and computed Cliff's Delta [30] (in a paired setting with $\alpha = .05$) as effect size. In both analyses, the two groups compared were the per-vulnerability-category F1 values across the 14 categories between each pair of tools. We chose these analyses as they are non-parametric hence make no assumption about the normality of the distribution of underlying data points. Given a Cliff's Delta value d, we interpret the effect size as follows [31]: effect size is *negligible* if $|d| \leq 0.147$, *small* if $0.147 < |d| \leq 0.33$, *medium* if $0.33 < |d| \leq 0.474$, and *large* if $|d| > 0.474$.

III. RESULTS

We present major findings with respect to each of our three research questions separately as follows.

A. RQ1: Effectiveness

Table I shows the recall, precision, and F1 accuracy (first row) of each of the five detectors chosen (second row) against each of the 14 vulnerability categories studied (first column). The size of a category is the number of benchmarks of that category used in the study: each category contains equal numbers of positive and negative samples. The results indicate that **the detectors had mostly perfect precision**, except for ADDRESSSANITIZER and CBMC against the *stack underrun* vulnerabilities. Noting that all the vulnerabilities reported by purely dynamic analysis based tools were true positives, a plausible explanation for the exception is that **the static analysis involved introduced imprecision** for the particular type of vulnerabilities. This confirms that stack overrun is challenging to detect statically.

Looking at the recall, however, revealed that quite some of the perfect-precision numbers were a result of zero recall—where there was no true vulnerability captured, the precision was *trivially* perfect. Generally, **the recall was low with these detectors for many vulnerability categories**. CBMC had relatively the highest recall, intuitively because of the general recall advantage of static analysis. Similarly, ADDRESSSANITIZER had the second best recall since it incorporates a static analysis phase. With the perfect precision in general, this led to these two tools having the highest F1 accuracy as well. The reason that **dynamic analysis did not seem to bring accuracy benefits compared to static analysis for vulnerability detection here** is probably because these vulnerabilities were purposely curated (to be detectable) for static analysis tools as noted earlier (where dynamic detection is not so much needed).

Overall, the effectiveness of these detectors varied widely: 66.7% to 100% precision, 0% to 100% recall, and 0% to 100% F1 per category, indicating most of the techniques worked extremely well on certain kinds of vulnerabilities yet quite poorly on others. A plausible reason for this large variation is that the detection techniques tend to focus purposely on particular kinds of vulnerabilities—detecting a greater variety of vulnerabilities would be much more challenging. The bottom row of the table shows the overall effectiveness computed by treating the entire benchmark suite as a whole: The two detectors with static analysis, CBMC and ADDRESSSANITIZER, achieved the highest F1 accuracy (87.3% and 77.7%, respectively). VALGRIND was the best-performing purely dynamic tool (62.4% F1), while MEMORYSANITIZER performed the worst (40.5% F1).

B. RQ2: Efficiency

Table II lists the per-benchmark average time costs incurred by each of the studied detectors, for each vulnerability category (third to sixteenth rows) and over all the 520 benchmarks (last row). The format is similar to Table I, except

TABLE I: Effectiveness of the five compared memory vulnerability detectors, per category (size in parentheses) and overall

Vulnerability Category	Recall					Precision					F1				
	Valgrind	DrMemory	Address Sanitizer	Memory Sanitizer	CBMC	Valgrind	DrMemory	Address Sanitizer	Memory Sanitizer	CBMC	Valgrind	DrMemory	Address Sanitizer	Memory Sanitizer	CBMC
Dynamic Buffer Overrun (64)	100%	100%	100%	3.13%	100%	100%	100%	100%	100%	100%	100%	100%	100%	6.06%	100%
Dynamic Buffer Underrun (78)	92.31%	87.18%	61.54%	94.87%	100%	100%	100%	100%	100%	100%	96.00%	93.15%	76.19%	97.37%	100%
Static Buffer Overrun (108)	3.70%	3.70%	90.74%	3.70%	100%	100%	100%	100%	100%	100%	7.14%	7.14%	95.15%	7.14%	100%
Static Buffer Underrun (26)	0%	0%	61.54%	0%	100%	100%	100%	100%	100%	100%	0%	0%	76.19%	0%	100%
Stack Overflow (14)	14.29%	0%	85.71%	85.71%	0%	100%	100%	100%	100%	100%	25.00%	0%	92.31%	92.31%	0%
Stack Underrun (14)	85.71%	85.71%	100%	14.29%	57.14%	100%	100%	77.78%	100%	66.67%	92.31%	92.31%	87.50%	25.00%	61.54%
Invalid Memory Access Already Freed Area (34)	82.35%	82.35%	82.35%	17.65%	58.82%	100%	100%	100%	100%	100%	90.32%	90.32%	90.32%	30.00%	74.07%
Cross Thread Stack Access (12)	0%	16.67%	0%	0%	0%	100%	100%	100%	100%	100%	0%	28.57%	0%	0%	0%
Double Release (12)	0%	0%	0%	0%	0%	100%	100%	100%	100%	100%	0%	0%	0%	0%	0%
Double Free (24)	91.67%	91.67%	91.67%	0%	100%	100%	100%	100%	100%	100%	95.65%	95.65%	95.65%	0%	100%
Free Non-Dynamically Allocated Memory (32)	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Free Null Pointer (28)	0%	0%	0%	0%	0%	100%	100%	100%	100%	100%	0%	0%	0%	0%	0%
Data Overflow (50)	0%	0%	0%	0%	60.00%	100%	100%	100%	100%	100%	0%	0%	0%	0%	75.00%
Data Underflow (24)	0%	0%	0%	0%	66.67%	100%	100%	100%	100%	100%	0%	0%	0%	0%	80.00%
All (520)	45.38%	44.62%	64.23%	25.38%	78.08%	100%	100%	98.82%	100%	99.02%	62.43%	61.70%	77.86%	40.49%	87.31%

TABLE II: Efficiency of the five compared memory vulnerability detectors, per category (size in parentheses) and overall

Vulnerability Category	Positive Samples						Negetive Samples					
	Directly	Valgrind	DrMemory	Address Sanitizer	Memory Sanitizer	CBMC	Directly	Valgrind	DrMemory	Address Sanitizer	Memory Sanitizer	CBMC
Dynamic Buffer Overrun (64)	<0.1ms	1,018.8ms	578.8ms	26.8ms	26.8ms	290.4ms	<0.1ms	1,058.4ms	592.2ms	29.8ms	26.8ms	298.8ms
Dynamic Buffer Underrun (78)	<0.1ms	1,019.2ms	575.0ms	25.6ms	30.2ms	362.6ms	0.2ms	1,052.0ms	577.2ms	29.4ms	27.2ms	343.8s
Static Buffer Overrun (108)	<0.1ms	1,020.6ms	568.7ms	27.1ms	27.4ms	214.7ms	<0.1ms	1,021.8ms	591.9ms	30.7ms	27.6ms	208.9s
Static Buffer Underrun (26)	2.5ms	934.1ms	506.1ms	29.8ms	27.8ms	139.4ms	<0.1ms	1,019.4ms	626.1ms	30.8ms	27.2ms	139.7ms
Stack Overflow (14)	<0.1ms	993.7ms	573.7ms	33.1ms	28.4ms	139.4ms	<0.1ms	1,022.3ms	573.1ms	33.1ms	26.7s	**2,0875.4ms**
Stack Underrun (14)	0.6ms	1,022.9ms	4,948.5ms	26.9ms	30.7ms	**21,775.4ms**	<0.1ms	1,029.7ms	669.7ms	28.0ms	28.4ms	1,187.4ms
Invalid Memory Access to Already Freed Area (34)	0.2ms	1,023.1ms	572.0ms	26.8ms	28.1ms	366.6ms	<0.1ms	1,084.9ms	575.1ms	32.0ms	27.8ms	350.1ms
Cross Thread Stack Access (12)	<0.1ms	1128.6ms	614.7ms	32.0ms	30.3ms	250.0ms	<0.1ms	1132.0ms	652.7ms	32.0ms	29.0ms	244.7ms
Double Release (12)	<0.1ms	1,085.3ms	630.7ms	35.3ms	26.0ms	229.3ms	1.3ms	1,084.7ms	643.3ms	32.0ms	30.3ms	229.3ms
Double Free (24)	0.3ms	1,007.7ms	571.3ms	28.3ms	26.6ms	166.0ms	0.3ms	1,017.3ms	575.0ms	32.7ms	28.3ms	164.7ms
Free Non-Dynamically Allocated Memory (32)	<0.1ms	1,071.5ms	716.75ms	31.0ms	30.7ms	183.5ms	<0.1ms	1,014.7ms	615.0ms	32.0ms	27.0ms	183.5ms
Free Null Pointer (28)	<0.1ms	1,016.3ms	550.3ms	25.7ms	27.6ms	321.1ms	<0.1ms	1,105.7ms	573.1ms	28.3ms	29.3ms	456.0ms
Data Overflow (50)	<0.1ms	1,020.6ms	627.5ms	28.9ms	27.6ms	133.8ms	<0.1ms	1,016.2ms	618.6ms	29.1ms	27.6ms	137.6ms
Data Underflow (24)	0.3ms	1,013.0ms	567.3ms	31.3ms	29.0ms	130.0ms	0.3ms	1,122.7ms	592.3ms	31.3ms	29.0ms	132.0ms
All (520)	<0.1ms	1,021.3ms	702.5ms	28.2ms	28.1ms	822.4ms	<0.1ms	1,047.6ms	597.8ms	30.4ms	28.2ms	829.8ms

TABLE III: Statistic significance and size of F1 accuracy differences between each pair of the studied detectors

Pair of detectors	p value	effect size
VALGRIND-DRMEMORY	1	-0.071
VALGRIND-ADDRESSSANITIZER	0.281	0.071
VALGRIND-MEMORYSANITIZER	0.247	-0.143
VALGRIND-CBMC	0.236	0.214
DRMEMORY-ADDRESSSANITIZER	0.402	0
DRMEMORY-MEMORYSANITIZER	0.205	-0.214
DRMEMORY-CBMC	0.236	0.214
ADDRESSSANITIZER-MEMORYSANITIZER	**0.035**	-0.357
ADDRESSSANITIZER-CBMC	0.635	0.214
MEMORYSANITIZER-CBMC	**0.032**	**0.571**

for that we show the costs for positive and negative samples separately—we intended to see if the presence/absence of vulnerabilities was correlated with higher/lower analysis costs. Each cost number included all relevant parts of the time spent (e.g., recompiling the benchmark if necessary). To help understand the run-time (slow-down) overheads incurred by *run-time tools*, the table (second and eighth columns) also lists the average original execution time per benchmark (*Directly*).

The numbers show that **these detectors were generally extremely fast against the benchmarks used, costing about one second per benchmark in most cases**. The reason is likely because these benchmarks are mostly simple, short programs—which also explains the negligible execution time of the original programs (i.e., the *Directly* column). The most costly cases (21 seconds), as highlighted in boldface, were seen by CBMC against the *stack underrun* positive samples and *stack overflow* negative samples. Our manual inspection revealed that the reason was because the relevant benchmarks include loops and recursions, which are expensive to analyze statically. Two dynamic detectors, ADDRESSSANITIZER and MEMORYSANITIZER, were peculiarly fast, mainly because of the lightweight nature of their analysis when built on top of the LLVM framework. The efficiency differences among the other three detectors were small. Also, for any detector, there were no significant differences in efficiency between positive and negative samples, and **there was no consistent correlation between the efficiency and the sample being positive or negative**. On the other hand, the run-time overheads incurred by dynamic tools were substantial in terms of slowdown (as percentages), albeit not very much significant practically in terms of the (small) absolute cost numbers.

C. RQ3: Comparison

The results of our two statistical analyses are shown in Table III, where the cases with statistical significance or *large* effect size are highlighted in boldface. The numbers indicate that only CBMC was significantly more accurate than MEMORYSANITIZER with a statistically large difference in terms of the F1 measure. This is consistent with the foregoing observation that CBMC had the highest, while MEMORYSANITIZER had the lowest, accuracy according to our effectiveness results (Table I. Between MEMORYSANITIZER and ADDRESSSANITIZER, the accuracy difference was significant but not large (only with a medium effect size of 0.357): the negative sign means the accuracy of the second was lower than that of the first in respective pairs. In all other cases, the detectors contrasted were not significantly different in vulnerability detection accuracy, with respect to the benchmarks considered at least.

In sum, the only **purely static detector was significantly more accurate than the weakest purely dynamic detector**. This is again possibly a result of the fact that the benchmarks were mainly designed for evaluating static analysis tools [21].

D. Case Studies

We have finished 13 of our case studies for understanding the reasons behind the successes and failures of particular

detectors against particular categories of vulnerabilities. From these case studies, we found that generally **dynamic tools had better performance on heap and stack vulnerabilities while static tools performed better against vulnerabilities due to syntactic coding errors**. Due to space limit, we only elaborate on one case below, which examines why VALGRIND had poor recall on our *static buffer overrun* benchmarks.

```
1 void first ()
2 {
3         char buf[5];
4         buf[5] = 1; /*Tool should detect this line as error*/
5         ......
6 }
7 void second ()
8 {
9         int buf[5];
10        int index;
11        index = rand();
12        buf[index] = 1; /*Tool should detect this line as error*/
13        ......
14 }
```

Listing 1: Snippets of two *static buffer overrun* benchmarks.

Listing 1 shows part of two benchmarks in this category, of which VALGRIND detected the vulnerability in the second but did not in the first, although the first seems easier to detect. The reason is because this detector, like DRMEMORY, is a purely dynamic detector, thus it **did not check the static buffer size in code**. Instead, it **only checked whether the address being visited was valid in the stack or heap**. In the first benchmark, `buf[5]` was still found as a valid memory block within the stack of the program, thus the vulnerability was missed. In the second, however, `rand()` returned a large number far beyond the size of the valid stack region of the program, thus the vulnerability was captured at runtime. Among the 52 positive samples in this category, only two were like the second benchmark and all others like the first. Thus, only the two out of 52 were successfully detected, hence the 3.7% recall.

IV. CONCLUSION AND FUTURE WORK

As part of an ongoing study, we evaluated five memory vulnerability detectors against a public C/C++ benchmark suite. We found that the static detector chosen performed the best in terms of detection accuracy (mainly due to its higher recall), while two dynamic detectors were faster than others (due to the underlying framework's efficiency). Yet there were no statistically significant and large accuracy differences between most of these detectors. By considering more detectors and a much larger and more diverse set of benchmarks (mainly real-world programs), we are expanding our empirical study. We are also conducting more in-depth case studies to obtain more generalizable understandings about the reasons behind varied tool performance so as to distill practical and actionable recommendations for future vulnerability defense development.

V. ACKNOWLEDGMENTS

We thank the reviewers for their insightful feedback. This work is supported in part by NSF grant CCF-1936522.

REFERENCES

[1] Tricentis, "Software fail watch: 5th edition," Tricentis, Tech. Rep., March 2017, https://www.tricentis.com/resources/software-fail-watch-5th-edition/.
[2] B. Arkin, S. Stender, and G. McGraw, "Software penetration testing," *IEEE Security & Privacy*, vol. 3, no. 1, pp. 84–87, 2005.
[3] A. Austin, C. Holmgreen, and L. Williams, "A comparison of the efficiency and effectiveness of vulnerability discovery techniques," *Info. and Soft. Technology*, vol. 55, no. 7, pp. 1279–1288, 2013.
[4] H. Kim, T.-H. Choi, S.-C. Jung, H.-C. Kim, O. Lee, and K.-G. Doh, "Applying dataflow analysis to detecting software vulnerability," in *Intl. Conf. on Advanced Communication Technology*, 2008, pp. 255–258.
[5] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in Java applications with static analysis." in *USENIX Security Symposium*, vol. 14, 2005, pp. 18–18.
[6] W. Du and A. P. Mathur, "Vulnerability testing of software system using fault injection," *Purdue University, Technique Report*, pp. 98–02, 1998.
[7] B. Chess and J. West, "Dynamic taint propagation: Finding vulnerabilities without attacking," *Information Security Technical Report*, vol. 13, no. 1, pp. 33–39, 2008.
[8] G. Jie, K. Xiao-Hui, and L. Qiang, "Survey on software vulnerability analysis method based on machine learning," in *International Conference on Data Science in Cyberspace*, 2016, pp. 642–647.
[9] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, p. 56, 2017.
[10] A. Austin and L. Williams, "One technique is not enough: A comparison of vulnerability discovery techniques," in *Intl' Symposium on Empirical Software Engineering and Measurement*, 2011, pp. 97–106.
[11] D. Pozza, R. Sisto, L. Durante, and A. Valenzano, "Comparing lexical analysis tools for buffer overflow detection in network software," in *Intl' Conf. on Communication Systems Soft. & Middleware*, 2006, pp. 1–7.
[12] N. Antunes and M. Vieira, "Comparing the effectiveness of penetration testing and static code analysis on the detection of SQL injection vulnerabilities in web services," in *Pacific Rim International Symposium on Dependable Computing*, 2009, pp. 301–306.
[13] P. Li and B. Cui, "A comparative study on software vulnerability static analysis techniques and tools," in *International Conference on Information Theory and Information Security*, 2010, pp. 521–524.
[14] R. Amankwah, P. K. Kudjo, and S. Y. Antwi, "Evaluation of software vulnerability detection methods and tools: A review," *Intl' Journal of Computers and Applications*, vol. 169, no. 8, pp. 22–27, 2017.
[15] P. Silberman and R. Johnson, "A comparison of buffer overflow prevention implementations and weaknesses," *IDEFENSE, August*, 2004.
[16] N. Antunes and M. Vieira, "On the metrics for benchmarking vulnerability detection tools," in *DSN*, 2015, pp. 505–516.
[17] ——, "Benchmarking vulnerability detection tools for web services," in *International Conference on Web Services*, 2010, pp. 203–210.
[18] A. Tajpour and M. J. zade Shooshtari, "Evaluation of SQL injection detection and prevention techniques," in *Intl' Conf. on Computational Intelligence, Communication Systems and Networks*, 2010, pp. 216–221.
[19] J. Fonseca, M. Vieira, and H. Madeira, "Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks," in *Pacific Rim Intl' Symp. on dependable computing*, 2007, pp. 365–372.
[20] A. Tajpour, M. Massrum, and M. Z. Heydari, "Comparison of SQL injection detection and prevention techniques," in *Intl' Conf. on Education Technology and Computer*, vol. 5, 2010, pp. V5–174.
[21] S. Shiraishi, V. Mohan, and H. Marimuthu, "Test suites for benchmarks of static analysis tools," in *International Symposium on Software Reliability Engineering Workshops*, 2015, pp. 12–15.
[22] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007, pp. 89–100.
[23] D. Bruening and Q. Zhao, "Practical memory checking with Dr. Memory," in *CGO*, 2011, pp. 213–223.
[24] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *USENIX Annual Technical Conference*, 2012, pp. 309–318.
[25] E. Stepanov and K. Serebryany, "MemorySanitizer: fast detector of uninitialized memory use in C++," in *CGO*, 2015, pp. 46–55.
[26] D. Kroening and M. Tautschnig, "CBMC–C bounded model checker," in *TACAS*, 2014, pp. 389–391.
[27] A. Rasool, "Which is the most vulnerable programming language?" https://www.digitalinformationworld.com/2019/03/searching-for-the-most-secure-programming-language.html, 2019.
[28] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, p. 75.
[29] R. E. Walpole, R. H. Myers, S. L. Myers, and K. E. Ye, *Probability and Statistics for Engineers and Scientists*. Prentice Hall, 2011.
[30] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 1996.
[31] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and cohen's d indices the most appropriate choices," in *annual meeting of the Southern Association for Institutional Research*. Citeseer, 2006, pp. 1–51.